2005

Hardware and Software Multi-precision Implementations of Cryptographic Algorithms

Muhammad A. Janjua

Hardware and Software Multi-precision Implementations of Cryptographic Algorithms

by

Muhammad Ali Janjua

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

Supervised by

Dr. Marcin Łukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY

June, 2005

Approved By:

Dr. Marcin Łukowiak

Primary Advisor – R.I.T. Dept. of Computer Engineering

June 20/05

Dr. Stanisław P. Radziszowski

Secondary Advisor – R.I.T. Dept. of Computer Science

Dr. Muhammad Shaaban

Secondary Advisor – R.I.T. Dept. of Computer Engineering

Thesis/ Dissertation Author Permission Statement

Title of Thesis: Hardware and Software Multi-precision Implementations of Cryptographic Algorithms

Name of Author: Muhammad Ali Janjua
Degree: Master of Science
Major: Computer Engineering
College: Kate Gleason College of Engineering

As per current Rochester Institute of Technology (RIT) guidelines for completion of my degree, I understand that I need to submit a copy of my Master's thesis to the RIT Archives. I hereby permit RIT and its agents to archive and make use of my thesis or dissertation in whatever forms necessary. I retain the ownership rights to the copyright of the thesis or dissertation and also retain the rights to use all or part of my thesis in my future work.

Author signature Date 06/21/05

Acknowledgement

I would like to offer deep and sincere thanks to God the Gracious and Merciful, who blessed me with the capabilities of heart and mind, which lead me to successfully complete this thesis.

I am indebt to my family members for giving me courage and confidence with all their hopes and prayers for me.

Now as I submit my thesis, I express me heartiest gratitude to all those who helped me in completing my thesis.

Working on this thesis has been an experience, and education in its own way. For the first time in my entire education career I was able to work thoroughly within a challengeable environment which made me to learn more from industrial point of view.

My deepest gratitude is for Dr. Marcin Łukowiak, my thesis advisor, who was a guiding light for me. I found his expertise spread equally over theory as well as practical aspects. Specially, he guided my way to overcome problems occurring from time to time during the course of completion of this thesis.

I would like to thank the members of my thesis committee, Dr. Stanisław Paweł Radziszowski and Dr. Muhammad Shaaban, for taking the time from their busy schedule for advising me about the related questions, reviewing my work, and perfecting me for the final defense.

Finally, I would like to thank all of my professors with whom I took various courses which helped me in thinking creatively and dynamically to bring up the success to my thesis. I am thankful my supervisor during my internship, Mr. Jim O'Connor for giving me the confidence of learning SystemC, which I used in my thesis.

Abstract

The software implementations of cryptographic algorithms are considered to be very slow, when there are requirements of multi-precision arithmetic operations on very long integers. These arithmetic operations may include addition, subtraction, multiplication, division and exponentiation.

Several research papers have been published providing different solutions to make these operations faster. Digital Signature Algorithm (DSA) is a cryptographic application that requires multi-precision arithmetic operations. These arithmetic operations are mostly based upon modular multiplication and exponentiation on integers of the size of 1024 bits. The use of such numbers is an essential part of providing high security against the cryptanalytic attacks on the authenticated messages. When these operations are implemented in software, performance in terms of speed becomes very low. The major focus of the thesis is the study of various arithmetic operations for public key cryptography and selecting the fast multi-precision arithmetic algorithms for hardware implementation. These selected algorithms are implemented in hardware and software for performance comparison and they are used to implement Digital Signature Algorithm for performance analysis.

Table of Contents

List of Figures

List of Tables

Glosary

DSS: Digital Signature Standard

DSA: Digital Signature Algorithm.

RSA: A public key cryptosystem named after its inventers, Rivest, Shamir, and Adlerman.

SHA-1: 160 bit secure hashing standard.

GCD: Greatest Common Divisor.

ASIC: Application Specific Integrated Circuit.

VHDL: Very high speed integrated circuit Hardware Description Language.

Key: A number or equivalent representation form which is used to encrypt or decrypt information.

Primality Testing: Any algorithm used to verify a number if prime or not.

CPU: Central Processing Unit.

FSM: It stands for Finite State Machine. It is a control block to control the hardware data flow logic.

Synthesis: Conversion of code written in VHDL to describe actual behavior of hardware into gate level model. Gate level model represents logical hardware components, which can be physically implemented after place and route operation.

Place and Route: In place and route operation, the gate level model generated by synthesis operation are connected together using interconnects or wires to form a hardware design. This operation also generates SDF file, which contains the actual timing detail of the gate level model.

SDF: Standard delay format.

FPGA: It stands for field programmable gate array. It is used to implement the design, which has been processed through place and route operation. After place and route operation, a programming file is generated to be emulated on FPGA.

Chapter 1
Introduction

Application Specific Integrated Circuits (ASICs) have outperformed many applications running on general purpose processors in terms of speed. This is due to the nature of hardware implementation of these applications which gives the advantages of hardware parallelism, pipelining, dedicated resources and short length of data transfer.

Multi-precision cryptographic applications require use of very long numbers. i.e. Digital Signature Algorithm requires modular exponentiation of the size of 1024 bits. As a result, the software based application execution on a 32 bit general purpose processor becomes very slow because of sequential data operations, longer interconnects and lack of data parallelism. Implementation of dedicated hardware can remove these speed bottlenecks, and as a result a combination of hardware and software provides better performance. This combination can also be further optimized into System on Board or System on Chip to achieve more speedup.

Multi-precision modular arithmetic is an essential part of public key cryptosystems. Ordinary implementation of 1024 or 2048 bit modular arithmetic in hardware can affect the speed, because it requires use of division operation. Because of this, software based fast multi-precision algorithms cannot be implemented in hardware. The requirement of the long numbers is required to provide high security against cryptanalytic attacks on the authenticated messages.

1.1 Scope of Research

The research area covered in this thesis to recognize the speed bottlenecks in the software based multi-precision cryptographic algorithms. In order to remove these bottlenecks, fast multi-precision algorithms for modular multiplication and exponentiation are analyzed and implemented in hardware. The hardware is then further used to implement Digital Signature Algorithm (DSA) cryptosystem. The implementation of DSA has been done in both software and hardware for performance comparison. The hardware implementation is particularly targeted for those portions of DSA where multi-precision modular exponentiation has been used.

1.2 Organization of Thesis

Chapter 2 is provides a brief introduction about cryptography and the use of multi-precision arithmetic.

Chapter 3 then further elaborates the requirement of multi-precision arithmetic by describing the most commonly used cryptographic algorithms.

Chapter 4 covers the selected Modular arithmetic methods, their hardware implementation and analysis.

Chapter 5 provides the implementation of Digital Signature Algorithm while using the selected algorithms in chapter 4. Chapter 5 also gives hardware and software comparison for speed.

Chapter 6 includes the conclusions and future work.

Chapter 2
Cryptography, an introduction

Information security has been a major concern for many years. Several techniques have been developed to secure the information over short and long range communication. This concept of securing information belongs to the field of cryptology. Generally, cryptology is the field of study which provides information about the communication of data over non-secure channels. It is further subdivided into cryptography and cryptanalysis. Cryptography is the process of designing systems to secure information where as cryptanalysis is the process of breaking those systems. In cryptographic science cipher text is a term used for the encrypted information.

2.1 Cryptography

"*The mathematical science used to secure the confidentiality and authentication of data by replacing it with a transformed version that can be reconverted to reveal the original data only by someone holding the proper cryptographic algorithm and key.*" [1]

From ancient cryptographic systems the first recorded use of cryptography found is by the Spartans who (400 BC) employed a cipher device called a "scytale" to send secret communications between military commanders. The scytale consisted of a tapered stick around which was wrapped a piece of parchment inscribed with the message. Once unwrapped the parchment appeared to contain an incomprehensible set of letters, however when wrapped around another stick of identical size the original text appears. There are many examples of ancient cryptographic systems available these days. Julius Caeser often used a simple cipher, which was later named after him, "Caeser Cipher". His method of encryption and decryption was based upon shifting of letters by three spaces.

With the growing needs to have fast and secure communication, the old cryptographic systems can only be referenced to a small fraction of what is used these days. As the distance between an encrypter and decrypter becomes longer, the security becomes more critical when transferring some important information. Demand of more security increases the complexity of systems, but for longer distances, the design of

complex systems become difficult. The data encryption and decryption of a message is done by using key. A key is a number or equivalent representation form which is used to encrypt or decrypt information. Modern cryptography is categorized into two major types, asymmetric key cryptography, and symmetric key cryptography.

Symmetric key cryptography is commonly termed as secret key or private key cryptography. Private key cryptosystems use same key for both encryptions and decryptions. They are mathematically less complicated than asymmetric key cryptosystems. Security in private key cryptosystems requires secure channels of communication. Generally the security also depends upon, how the key is exchanged between the two parties. This exchange is usually done by using asymmetric key methods. These cryptosystems are usually implemented for shorter distance of information exchange. These cryptosystems do not require complex multi-precision arithmetic, thus they will be ignored as part of research.

Asymmetric key cryptography is also commonly termed as public key cryptography. Its concept was introduced in 1970. The word public key is based upon the concept, that anyone can retrieve a key either for encryption or decryption but not for both. In this case, there are thus two keys, one is considered public key, and one is secret. One of a use of such cryptosystems is for long distance information exchange. Fast cryptanalytic attacks have made these systems unsecured, thus there is always an improvement required with the growth of fast systems.

Consider the basic structure of public key cryptosystem, which is divided into an encrypter E, and a decrypter D. Suppose E wants to send an encrypted message to D, which D then decrypts, but they are located very far from each other. Due to longer distances they cannot agree upon any key to use because they cannot meet each other. The concept of public key then arises here, that, the decrypter sends or broadcast a public key and either one encrypter or more encrypt their messages to form cipher texts. The cipher texts then goes back to the decrypter, who already has the inverse of the public key, which he uses to decrypt the cipher. The inverse of the public key is kept secret. The following diagram describes this concept,

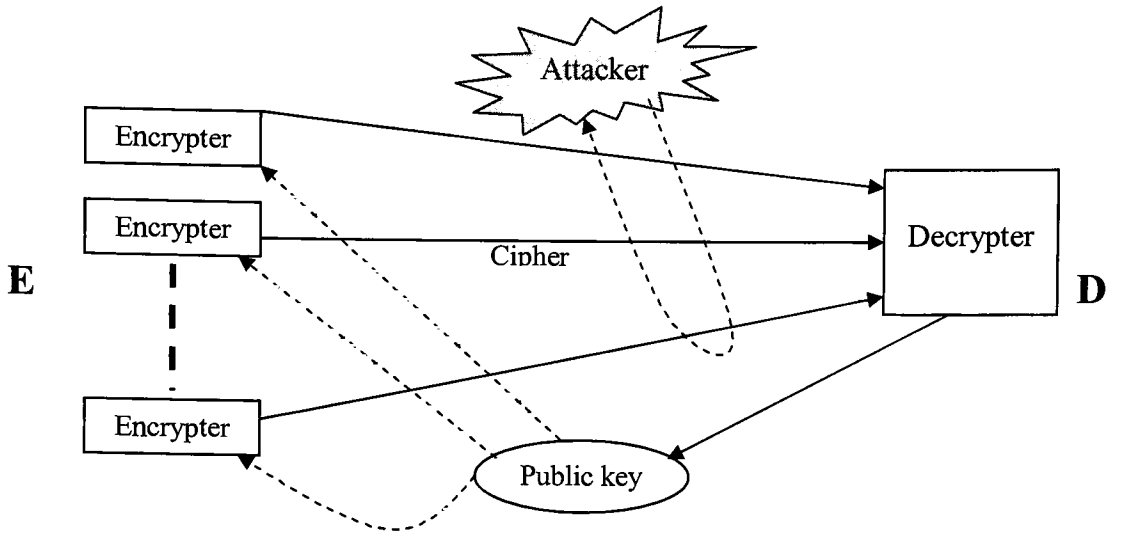


Figure 2.1: Public Key Cryptosystem

In the figure 2.1 an attacker is also shown besides *n* number of encrypters and decrypter. There can be several goals of the attacker, the common ones are

a) To read the message.
b) Retrieving the secret key, and thus reading all publicly encrypted messages.
c) Corrupting an original cipher text into another cipher so that the decrypter gets a wrong cipher text.
d) Dodge the decrypter by pretending as a valid encrypter.

There are four possible types of attacks that the attackers can perform on the cipher text,

a) **Cipher text only attack:** The attacker is only able to copy the cipher text, and then he can try different ways to decrypt the cipher text.
b) **Known plaintext attack:** It is more destructive attack then the cipher text only attack. The attacker gets the copy of cipher text as well the original corresponding text. Then he can deduce the key out of it. Once he has the secret key, and the decrypter doesn't change the secret key so the attacker is able to read all the messages in future.
c) **Chosen plaintext attack:** Using this method, if the attacker has access to the actual encryption machine, he encrypts many chosen plain texts to deduce the secret key out of it.

d) **Chosen cipher text attack:** In this method the attacker gains an access to the decrypter machine and tries to decrypt many symbols in order to deduce the key.

In order to secure the information many cryptosystems has been developed. Digital Signature Algorithm is a cryptosystem which is used to secure the document authentication. Next section describes in detail about Digital Signature Algorithms.

2.1.1 Digital Signatures

A signature represents the authenticity of a document with its association to the signer. Presently signatures have attained much importance in the digital media, especially when the distance between the message signer and signature verifier increases.

When authentication of a document is considered then an intruder may try to steal the authentication of the document by copying the original signature. Electronically same situation exists, and thus the signed document needs to be encrypted and then to be verified.

Digital signatures algorithms are used for signing the documents and verifying. There are two basic techniques usually considered for digital signatures. One is RSA signatures and the other is ElGamal signature scheme. Digital signatures are part of public key cryptography.

2.1.1.1 RSA Digital Signature

RSA signature [3] uses the same concept of RSA public key cryptosystem [3]. First consider two persons, Alice and Bob, where Alice is the signer and Bob is the verifier.

Bob sends a document m to Alice to sign it. The algorithm begins as follows,

1. Alice finds two primes p and q and then computes $n = pq$. She next chooses e_A such that $1 < e_A < \phi(n)$ with gcd $(e_A, \phi(n)) = 1$, and calculates d_A such that $e_A d_A \equiv 1 \pmod{\phi(n)}$. Where $\phi(n) = (p-1)(q-1)$. Alice publishes (e_A, n) and keeps private d_A, p and q.
2. Alice's signature is

$$y \equiv m^{d_A} \pmod{n} \tag{2.1}$$

3. The pair (m, y) is then made public as a signed message.

Bob downloads the pair (m, y) and the public key (e_A, n). Bob verifies the signature by using the following steps,

1. Calculate $z \equiv y^{e_A} \pmod{n}$. If $z = m$, then Bob accepts the signature as valid; otherwise the signature is not valid.

2.1.1.2 ElGamal Digital Signature Scheme

ElGamal signature scheme [3] is based upon ElGamal encryption. A difference from RSA is that with ElGamal method, there can be many different signatures produced that are valid for a given message. Based upon ElGamal encryption method, the National Institute of Standards and Technology (NIST) proposed the Digital Signature Standard [17] in 1991 and adopted it as a standard in 1994.

2.1.1.3 Digital Signature Algorithm

Digital Signature Algorithm (DSA) [3] based upon Digital Signature Standard requires a hashed message SHA(m), which is 160-bit long. This hashed message is then signed and verified in this algorithm. Before explaining the algorithm in detail, first consider a hash algorithm SHA, which has been used in the design. The current standard for hash algorithm is Secure Hash Standard Algorithm [7] named as SHA-1. It was developed by NIST along with the NSA for use with the Digital Signature Standard (DSS) and is specified within the Secure Hash Standard (SHS) National Institute of Standards and Technology. SHA-1 is a cryptographic message digest algorithm which takes a message $m < 2^{64}$ bit plain text and converts it into 160 bit hashed message SHA(m). This 160 bit hashed message SHA(m) is called as a message digest. It is designed to have the following properties: it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest [8]. i.e. if there is a hash for document $m1$, it is difficult to find a document $m2$ which has the same hash. The algorithm of SHA-1 is given in [7], and that's why the algorithm will not be explained here. For the thesis research, the implementation of SHA-1 was done using multi-precision C++. A generated

hashed message SHA(m) is then used in DSA. Figure 2.2 describes the use of DSA along with SHA-1 for both signing and verification procedures.

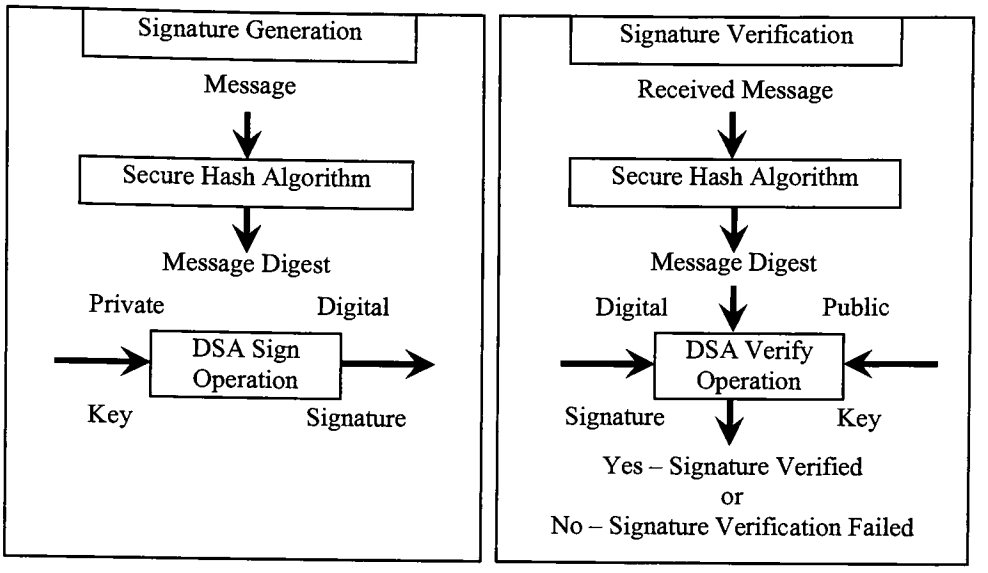


Figure 2.2: Data flow diagram of Digital Signature Algorithm using SHA-1

In figure 2.2, a hash function is used in the signature generation process to obtain a data, called a message digest. The message digest is then used by DSA to generate the digital signature. The digital signature is sent to the verifier in terms of signed message and public key. The verifier verifies the signature by using the signer's public key. The same hash function must also be used in the verification process. For this thesis research, the hash function is only used at the beginning of signing operation, because the purpose of the research is to explain the difference between hardware and software implementation of multi-precision arithmetic algorithms. DSA is used to implement these multi-precision algorithms for performance comparisons between hardware and software. A block diagram of a modified version of DSA is mentioned in the figure 2.3. A common SHA-1 block is used by both of the signature generation and sign verification blocks. The signature generation block generates a public key and a signed message for the verification process. In verification process, the signed message is verified using the hashed message.

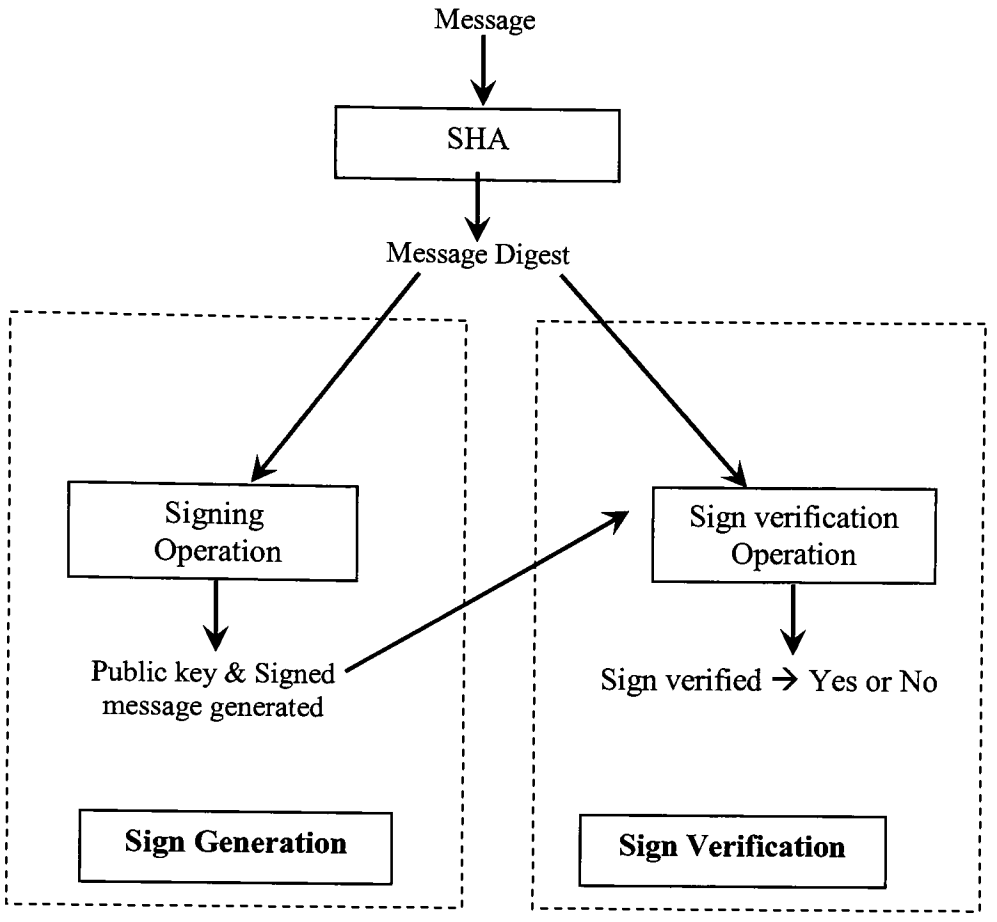


Figure 2.3: Block diagram Digital Signature Algorithm used for the thesis research.

Consider Alice as a Signer and Bob as a Verifier. Bob has a message digest SHA(m) which is 160 bits long, and he wants Alice to sign the message. Alice signs the message, and sends it back to Bob along with a public key. Bob uses the public key to verify the authenticity of the signature. Consider the following algorithm as mentioned in [3], page 190.

2.1 Algorithm DSA

Alice creates the public keys as follows,

1. Alice finds a prime q as a 160 bit long and chooses a prime p that satisfies $q \mid p\text{-}1$. i.e. $p = qx + 1$ is also prime. x is the even factor of the p-1. The discrete log problem should be hard for this choice of p.
2. Let g be a primitive root mod p and let,

$$\alpha \equiv g^{(p-1)/q} \pmod{p}, \tag{2.2}$$

Then, $\alpha^q \equiv 1 \pmod{p}$ must be satisfied.

3. Alice chooses a secret a such that $1 \leq a < q\text{-}1$ and calculates

$$\beta \equiv \alpha^a \pmod{p} \tag{2.3}$$

4. Alice publishes (p, q, α, β) as public key, and keeps a secret.

Next Alice signs the message using $\alpha, p, q, a,$ *and* SHA(m).

1. Alice selects a random, secret integer k such that $0 < k < q - 1$.

   She then computes

$$r \equiv (\alpha^k \pmod{p}) \pmod{q} \tag{2.4}$$

2. Next she computes

$$s \equiv k^{-1} (\mathrm{SHA}(m) + ar) \pmod{q} \tag{2.5}$$

3. Alice's signature is (r,s), which she sends to Bob.

On the other side, Bob receives the data including the public key and the signed message.

1. Bob first downloads Alice's public key information (p, q, α, β).
2. He computes,

$$u_1 \equiv s^{-1} \mathrm{SHA}(m) \pmod{q} \tag{2.6}$$

   and,

$$u_2 \equiv s^{-1} r \pmod{q} \tag{2.7}$$

3. Next he computes

$$v \equiv (\alpha^{u1} \beta^{u2} \pmod{p}) \pmod{q} \tag{2.8}$$

4. Bob accepts the signature, if and only if

$$v = r. \tag{2.9}$$

2.2 Summary

After a brief review of RSA and DSA public key cryptosystems, it is evident that the modular operation is the important part of the security. In 1024 bit RSA and DSA, the arithmetic operations involving exponents for modular operations require lot of computation cycles. For example in 1024 bit DSA cryptosystem, computation of cipher $\beta \equiv \alpha^a \pmod{p}$ requires 1024 bit data to be used. Performing software based multi-precision arithmetic operations on a single general purpose processor (GPP) is very slow because of lack of hardware parallelism and dedicated hardware resources. If dedicated hardware is used to replace the slower software blocks then a very efficient solution can be presented as a hardware and software combination. Such problems and their solutions are explained in this thesis report. Elgamal DSA has been targeted for efficient implementation and performance comparisons and the hardware implementation is shown in Chapter 5. The performance comparison has been done between the implementation of slower modules like modular exponentiation in software and hardware. The next chapter describes the fast multi-precision cryptographic algorithms.

Chapter 3

Multi-precision arithmetic in cryptography.

In modern cryptography, information is encoded and decoded in terms of numerical values. Information is first converted into numerical form using various conversion methods. The latest message conversion algorithms are secure hashing standard and Message Digest. They are usually based upon the principle of hashing the messages.

Once the information is converted into its numerical form m, it is then passed through the encryption algorithm. In encryption algorithm, certain arithmetic operations are done on m and produce a cipher c. This cipher c is when received on the decryption side; the decryption algorithm contains the arithmetic operations which converts c to m. This whole process may require from simple mathematical to very complex mathematical operations. As the cryptosystems are becoming more secure, their mathematical complexity is increasing and thus producing many challenges for the designers to implement it in most efficient way. Before going into the details of the cryptographic arithmetic, let's review the concepts of basic number theory used in them.

3.1 Congruence

In chapter 2 of "Introduction to Cryptography" [3], public key cryptosystems are briefly described to show the mathematical implementations for encryption and decryption. Modular arithmetic is used in these cryptosystems to obtain the results of various functions. It is also termed as congruence. It is defined as,

"Let a, b and n be integers with $n \neq 0$. We say that

$$a \equiv b \pmod{n} \tag{3.1}$$

if $a - b$ is a multiple (positive or negative) of n." [3]. Simply, if the difference $a - b$ is integrally divisible by a number n (i.e. $\frac{a-b}{n}$ is an integer), then a and b *are* congruent to n. The quantity a is some time called as base, and the quantity b is some time called as residue or remainder.

$a \equiv b \pmod{n}$ can also be written as, $a = b + nk$ for some integer k (positive or negative). An example can be considered as, $a = 19$ and $n = 8$ then,

$$19 \equiv 3 \pmod{8}, \text{ or } 19 = 3 + 2 * 8.$$

In this case $k = 2$, and $b = 3$.

Another example using $a = -12$, and $n = 7$,

$$-12 \equiv 37 \pmod{7}, \text{ or } -12 = 37 + (-7 * 7)$$

In this case, $k = -7$ and $b = 37$.

There are four propositions for congruencies as from [3],

Let a, b, c, n be integers and $n \neq 0$.

- $a \equiv 0 \pmod{n}$, *if and only if n/a.*
- $a \equiv a \pmod{n}$.
- $a \equiv b \pmod{n}$ *if and only if* $b \equiv a \pmod{n}$.
- *If* $a \equiv b$ *and* $b \equiv c \pmod{n}$, *then* $a \equiv c \pmod{n}$.

3.2 Greatest Common Divisor (GCD)

Considering a and b, the greatest common divisor [3] is the largest positive integer that divides both a and b. It is also called as greatest common factor and highest common factor. GCD is commonly used in public key cryptosystems particularly for finding relative primes. Relative primes are the two numbers, whose GCD is 1. Considering the following examples,

GCD (13, 21) = 1, GCD (2, 8) = 2, GCD (22, 71) = 1.

GCD of three or more than three numbers can be found by computing GCD of two numbers at a time, as,

GCD (a, b, c) = GCD (a, GCD (b, c))

Finding GCD has two methods:

1. Considering a and b as positive integers, the GCD can be computed by first factorizing the numbers into smallest primes, i.e.

$$a = \prod_i p_i^{\alpha i}$$

$$b = \prod_i p_i^{\beta i}$$

$1728 = 2^6 3^2$, $135 = 3^3 5$, GCD $(1728, 135) = 3^2 = 9$

2. If the numbers are very large, the factorization is very hard. The GCD can be calculated then by using Euclidean algorithm, which will be explained next.

3.3 Euclidean Algorithm

Euclidean algorithm [3] is generally used to find GCD of two integers. Suppose a and b are two integers, then there exits k and l such that,

$$ka + lb = c \tag{3.2}$$

where c is the GCD of a and b.

Algorithm 3.1: Euclidean-Algorithm (a, b)

1. $r_0 = a$; $r_1 = b$; $m = 1$;
2. while ($r_m \neq 0$) do

   2.1 $q_m = \left\lfloor \frac{r_{m-1}}{r_m} \right\rfloor$;

   2.2 $r_{m+1} = r_{m-1} - q_m r_m$;
3. $m = m+1$;

//r_m = GCD (a, b)

This algorithm only gives GCD of two integers. In order to find the multiplicative inverses of a and b, another version, known as extended Euclidean algorithm can be used.

3.4 Modular Exponentiation

Modular exponentiation [3] is very important part of the security of the most public key cryptosystems. Modern cryptosystems use very long numbers which causes arithmetic operations like modular exponentiation to operate at very slow speed. Considering an example, $2^{1027} \pmod{637} = 37$. If it is computed the power of 2 and reducing by 637, it will take 1027 iterations which is very slow operation. On the other hand the successive square on both sides and reduction of the base with modulus will reduce the number of iterations. Considering the above example,

Start with $2^2 \equiv 4 \pmod{637}$ and repeatedly square both sides to obtain the following congruencies:

$2^4 \equiv 4^2 \equiv 16$

$2^8 \equiv 16^2 \equiv 256$

$2^{16} \equiv 256^2 \equiv 562$

$2^{32} \equiv 529$

$2^{64} \equiv 198$

$2^{128} \equiv 347$

$2^{256} \equiv 16$

$2^{512} \equiv 256$

$2^{1024} \equiv 562$

Since the binary representation of 1027 = 10000000011

Then $2^{1027} \equiv 562 * 4 * 2 \equiv 37 \pmod{637}$. Where 562, 4, and 2 are selected based on logic 1's the binary representation of 1027. An efficient modular exponentiation algorithm is based upon right to left and left to right binary exponentiation algorithm.

3.4.1 Right-to-left binary exponentiation

The inputs to the Right-to-left binary exponentiation [12] are x and e, where x is the base and e is power.

Algorithm 3.2: Expo (x^e)

1. $A \leftarrow 1, S \leftarrow x$.
2. while ($e \neq 0$) do:
   - 2.1 if e is odd, then $A \leftarrow A \,.\, S$.
   - 2.2 $e \leftarrow \left\lfloor \frac{e}{2} \right\rfloor$.
   - 2.3 if $e \neq 0$, then $S \leftarrow S \,.\, S$.
3. return A.
4. End Expo (x^e).

In the above algorithm the loop runs until $e \neq 0$. At line 2.1, e is verified for if e is odd, then compute, $A \leftarrow A \,.\, S$. This verification of e to be odd or even can easily be checked using the LSB of e. If $e(0)$ = '1', then e is even, else it is odd. e is further divided by 2, which in hardware represents a right shift register, which when shifts right, the number is divided by 2. At line 2.3 $S \leftarrow S \,.\, S$ is computed. The multipliers at two lines

2.1 and 2.3 can be executed in parallel, if implemented in hardware although there is a dependency in the loop between line 2.1 and 2.3. i.e. S is updated at line 2.3 and then in the next iteration it is used by A at line 2.1. This dependency does not cause any conflict with the parallelism, because the multiplier at line 2.1 needs the resources of S to be used in the next iteration, but not within the same iteration.

3.4.2 Left-to-right binary exponentiation

The inputs to Left-to-right binary exponentiation [12] algorithm are x and e.

Algorithm 3.3: Expo (x^e)

1. $A \leftarrow 1$.
2. for $i =$ t down to 0, loop
   $A \leftarrow A \,.\, A$
   If $(e_i = 1)$, $A \leftarrow A \,.\, x$.
2. Return A.
3. End Expo (x^e).

In left-to-right binary exponentiation, the loop runs for time t, where t is the number of bits in e. When comparing to right-to-left algorithm, this algorithm has a data dependency within the same iteration, which does not allow achieving parallelism in this algorithm. This reduces the performance in terms of speed in this algorithm, although, the area requirements during the hardware implementation are half in left-to-right algorithm as compared to right-to-left algorithm. This is because the same multiplier can be used for both $A \leftarrow A \,.\, A$ and $A \leftarrow A \,.\, x$ iteratively.

In chapter 4, these two methods are further explained and one method is then chosen for hardware implementation.

3.5 Common Algorithms used in Cryptography

Most of the cryptosystems require complex computation based upon certain algorithms. These algorithms are helpful in providing fast computations and thus access to making cryptosystems more secure. The two algorithms used in this thesis research are

Extended Euclidean Algorithm and Primality test based upon Miller Rabin theorem. The description of these algorithms in chapter 3 signifies the multi-precision modular arithmetic used for public key cryptosystems. The purpose of showing the modular arithmetic is the hardware and software implementation of multi-precision arithmetic algorithms for performance comparison. The computation time in terms of hardware and software implementation varies, which is the target of this thesis research.

3.5.1 Extended Euclidean Algorithm

The Extended Euclidean algorithm [3] is a version of Euclidean algorithm. In addition to computing GCD, this algorithm also generates the multiplicative inverses of a and b only if a and b are relatively prime numbers.

Algorithm 3.4: Extended-Euclidean-Algorithm (a, b)

//a and b are positive integers

1. $r_0 = a$; $r_1 = b$;
2. $t_0 = 0$; $t_1 = 1$;
3. $s_0 = 1$; $s_1 = 0$;
4. $m = 1$;
5. while ($r_m \neq 0$) do
   - 5.1 $q_m = \left\lfloor \frac{r_{m-1}}{r_m} \right\rfloor$;
   - 5.2 $r_{m+1} = r_{m-1} - q_m r_m$;
   - 5.3 $t_{m+1} = t_{m-1} - q_m t_m$;
   - 5.4 $s_{m+1} = s_{m-1} - q_m s_m$;
   - 5.5 $m = m+1$;
6. $m = m-1$;

//r_m = GCD (a, b) and $s_m a + t_m b = r_m$ [4]

Due to the ability of this algorithm to provide the multiplicative inverses of the given two co-prime numbers, it is widely used for many cryptographic applications. As seen in chapter 2, RSA algorithm description, d has been computed as the multiplicative

inverse of e, which is the part of public key used for encryption of message. Besides finding multiplicative inverses of numbers, it also helps in avoiding the fractions created due to modulo division operation. The run time of this algorithm is O (n^2).

3.5.2 Primality Testing

Primality testing [3] is required in order to verify if a certain integer is prime or composite. For an integer N, the number of primes less than or equal to N is approximately N / ln N [4]. One property of prime numbers is that, they have no factors. Another is that every prime number is an odd number but not every odd number is a prime number. Numbers which are not primes are called as composite numbers. The term prime factorization refers to a prime p where (p -1) is an even number yields to its finite number of factors. When considering very long numbers, computational time of finding if a number is prime or not is less then prime factorization. Primality testing is now considered in polynomial time after M. Agrawal, N. Kayal, and N. Saxena, provided an algorithm that supposedly tested primality in polynomial time [5]. As compared to polynomial time algorithms of primality, the randomized or probabilistic algorithms are much faster and are widely used for Primality testing. Randomized algorithms are based upon *yes-based Monte Carlo algorithm* where a "yes" answer is always correct, but a "no" answer is probabilistic means, it can be incorrect. A most common probabilistic algorithm used for Primality is Miller-Rabin theorem [3].

Algorithm 3.5: Miller-Rabin (n)

1. define a, k, m, x_0, x_1 as integers.
2. choose a as a random integer.
3. $m = n - 1$; k=0;
4. while (m mod 2 = 0)do
   - 4.1 k= k+1;
   - 4.2 $m = m / 2$;
5. end while;
6. $x_0 \equiv a^m \pmod{n}$;
7. for i in 0 to k-1 loop

7.1 $x_1 \equiv x_0^2 \pmod{n}$;

7.2 if ($x_1 = 1$ and $x_0 \neq 1$ and $x_0 \neq n\text{-}1$) then return composite;

7.3 $x_0 = x_1$;

8. end loop;

9. if ($x_1 \neq 1$) then return composite;

10. return prime;

End Miller-Rabin(n).

Odd numbers are only tested for primes as mentioned above, so the input to this algorithm is only the odd number. The probability of Miller-Rabin test for a certain chosen random a is ¼ for a failure of recognizing a composite number.

3.6 Summary

The algorithms mentioned in this chapter are the basis of many cryptosystems. Modular exponentiation is an essential part of RSA and DSA public key cryptosystems. Chapter 4 describes the hardware implementation of modular multiplication and exponentiation algorithms. Miller-Rabin Primality test has been used as part of thesis research in order to find prime numbers for DSA. Extended Euclidean algorithm has been used to compute modular multiplication inverse of a secret random number in DSA.

Chapter 4
Hardware implementations of the multi-precision modular arithmetic methods

When considering highly secure cryptosystem, use of very long numbers is always targeted. Modern public key based cryptosystem like RSA and DSA normally uses numbers up to 2048 bits long. As presented in chapter 2, RSA and DSA algorithms requires modular exponentiation. When designing for speed, hardware replaces certain software portions which require selection suitable algorithms that fit the requirement of efficient hardware design.

Multi-precision modular exponentiation implementation in hardware gives much speedup when compared with software implementation. The speedup can be achieved by the availability of parallelism and dedicated hardware resources instead of a general purpose CPU which schedules the targeted cryptographic application tasks with other tasks. These are the most common consideration which causes the selection of dedicated hardware solution. If modular exponentiation is implemented in hardware using the division operation, it can consume lot of area and time resources on the hardware chip. This division can be avoided by using an algorithm for modular reduction which is explained next in the background work.

4.1 Background Work

a) Modular Multiplication without Trial Division [5]

In this paper Montgomery proposed a method for multiplying two integers modulo N while avoiding division by N. The representation of the integers is called as N. This method is useful only if several multiplications are performed for fixed modulus N. Fixed modulus is a drawback for such cryptosystems, where there is a requirement of using more than one modulus. Avoiding division is an essential part for hardware implementation, thus this algorithm has gained prime importance for the hardware implementation of multi-precision cryptography. This research has been used as the basis

of this thesis. Using Montgomery modular multiplication technique, an algorithm of modular exponentiation has been implemented in hardware.

b) Montgomery Modular Exponentiation on Reconfigurable Hardware [8]

In this paper, Montgomery modular multiplication was combined with systolic array design, which was capable of processing a variable number of bits per array cell. The design was targeted for Modular exponentiation which was further used as a design unit for 512 and 1024 bit RSA. The design was implemented on Xilinx XC4000 Series FPGA. The RSA decryption time for the 512 bit in hardware was 2.37ms as compared to the software implementation which was 9.1 ms on a 150 MHz Alpha. Thus the speedup in hardware was 3.8 times more as compared to software. Also the fastest 1024 bit software implementation of 43.3 ms running on a Pentium Pro–200 based PC was about 4 times slower than the hardware (10.2ms).

c) Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic. [6]

This paper presents a review of some existing architectures for the implementation of Montgomery modular multiplication and exponentiation on an FPGA. There are three different architectures of Montgomery multiplier described. These architectures are implemented, and comparison for area and speed is given. A comparison of two modular exponentiation algorithms using Montgomery multiplication is also described. A good selection of modular multiplier and exponentiator has been made to implement RSA cryptosystem. For this thesis research the algorithms presented in this paper have been chosen for hardware implementation. Instead of RSA, the selected architectures are implemented for DSA.

4.2 Classical Modular Reduction [3]

Consider two positive integers, x & y, and the modulus M. The representation of these numbers is in radix 2. The steps are,

a) Consider $x * y$, $x + y$, $x - y$, x, y modular reduction operations.

b) Compute remainder r for any of the arithmetic operations $x * y$, $x + y$, $x - y$, x, y divided by M.

c) return r.

In the steps above, the division is used in performing the modulo reduction operation. On the other hand let's consider Montgomery Modular reduction in this next section.

4.3 Montgomery Modular Reduction without trial division

Montgomery modular algorithm [5] avoids the division requirement in common modular multiplication, addition, subtraction, and single value reduction.
The basic algorithm is as follows,

Algorithm 4.1: Mont-reduce (x)

1. $u = (x \bmod R)\, M^{-1} \bmod R$
2. $t = \dfrac{x + uM}{R}$
3. if $t > M$ then
   3.1 $t = t - M$
4. return t

End Mont_reduce (x)

Inputs to the algorithm are x as the number to be reduced, M as the modulus, and R. The calculation of $R = 2^w$, where w is the number of bits in M. Also $0 < M < R$, and compute using Extended Euclidean Algorithm so that GCD (R, M) = 1, that is

$$RR^{-1} - MM^{-1} = 1 \tag{4.1}$$

This is equivalent to $M^{-1} = -M' \bmod R$. ($M' = \dfrac{1}{M}$),

for variable x to be reduce by modulo M, Montgomery algorithm uses

$$X \equiv x * R \bmod M, \tag{4.2}$$

Consider $M^{-1} \equiv -M' \bmod R$, and multiply both side by x

$$M^{-1} * x \equiv -M' * x \pmod R, \tag{4.3}$$

where $u = M^{-1} * x$

$$u \equiv -M' * x \bmod R, \tag{4.4}$$

multiply both side by M

$$u * M \equiv -(M' * M) * x \bmod R. \ (M'M = 1) \tag{4.5}$$

$$u * M \equiv -x \bmod R \tag{4.6}$$

$$u \equiv -M^{-1} * x \bmod R \tag{4.7}$$

$$u \equiv [(x \bmod R)(-M^{-1} \bmod R)] \bmod R, \tag{4.8}$$

where $M^{-1} \equiv -M' \bmod R$

$$u \equiv [(x \bmod R)(-1 * -M' \bmod R] \bmod R \tag{4.9}$$

$$u \equiv [(x \bmod R)(M' \bmod R] \bmod R. \tag{4.10}$$

This proves line 1.1 of Mont_reduce.

Now consider equation (4.6)→ $u * M \equiv -x \bmod R$

This is equivalent to,

$$u * M = -x + tR \tag{4.11}$$

for any constant t.

$$tR = x + u M \tag{4.12}$$

$$t = \frac{x + uM}{R} \tag{4.13}$$

or

$$t \equiv xR^{-1} \bmod M \tag{4.14}$$

This proves line 1.2 of Mont-reduce. The multiplication of x with R^{-1} and reduction by M gives a Montgomery residue form of x, which is t in this case. Montgomery residue form can also be obtained by multiplying x with R and reducing by M and getting X as in equation 4.2. Where $X \neq t$. Consider both of the cases for the reduction of x into original form y. i.e. $y \equiv x \pmod M$. If t is considered, then $y \equiv tR \pmod M$, otherwise for X, $y \equiv X R^{-1} \pmod M$. In the algorithm, the Modulus M is replaced by R, which is a shift register to shift the data serially towards right iteratively. One right shift is equivalent of dividing by 2.

In the algorithm, instead of dividing by M, R is used, which is 2^w. This in iterative implementation avoids the division by shifting the (x, M^{-1} in line 1), and, ($x+ uM$ in line 2) by 1 bit to the right.

Considering two operands A and B instead of one as x in the previous case, where $0 < A, B \leq M$. The properties for addition and multiplication then are,

1. $A + B$ is mapped to $(A + B)\,R = AR + BR$.
2. AB is mapped to $(A*B)\,R = AR\,(B*R)\,R^{-1}$

4.4 Montgomery Modular Multiplication without trial division [6]

When considering ordinary modular multiplication, two approaches are considered,

a) Performing modulo operation after multiplication.

b) Performing modulo operation during multiplication.

Modular operation requires using division, the remainder of which is used for further reduction. If the first approach is considered, it will require (w * w) bit multiplier, with a $2w$ bit register and ($2w$ * w) bit divider. w is the number of bits in the modulus. Using the second approach, the division can be avoided, but it will require more units of addition and subtractions to be involved. The ordinary modular multiplication algorithm for the computation of $(A*B)$ mod M takes the normal multiplication method which accumulates digit products A * b_i and interleaves modular reductions to keep the result below M. These reductions are achieved by subtracting the correct multiple of the modulus from the intermediate result. This reduction is dependent on the most significant bits of the operand. On the other hand, Montgomery in his algorithm for modular reduction [6] reverses the order of treating the digits of the multiplier by using the least significant bits of the intermediate result to perform an addition rather than a subtraction. A further shift down operation is then performed instead of a conventional algorithm shift up operation in each iteration.

$$A = \sum_{i=0}^{w-1} a(i)2^i, \qquad B = \sum_{i=0}^{w-1} b(i)2^i, \qquad M = \sum_{i=0}^{w-1} m(i)2^i$$

Algorithm 4.2: Mon_Pro (A, B, M) [6]

1. $A = 2A$; //(left shift by one bit)
2. $S_{-1} = 0$;
3. for $i = 0$ to n do
   3.1 $q_i = (S_{i-1}) \bmod 2$; //(q_i = LSB of S_{i-1})

3.2 $$Si = \frac{S_{i-1} + q_i M + b_i A}{2}$$

4. end for
5. return S_n
6. end Mon_Pro.

In the above algorithm, A, B, and M are inputs to the design. The loop runs for $n+1$ times, where $n = w+2$ and w is the number of bits in M. A is first shifted left. The additional two clock cycles are needed in order to keep the intermediate results in bound due to possible carry returns during the addition operations. An additional one clock cycle will be needed to reduce the effect of $2A$. Total number of $n+1$ clock cycles are needed to compute the final Montgomery product. This product P is obtained as,

$$p \equiv (A*B)\, R^{-1} \pmod{M} \tag{4.15}$$

R^{-1} is the multiplicative inverse of R (mod M) which is automatically included during the Montgomery multiplication operation.. In order to remove R^{-1}, $P \equiv p\, R \pmod{M}$ needs to be computed. This is also equal to,

$$P \equiv p\, R^{-1}\, R^2 \pmod{M} \tag{4.16}$$

Thus another Montgomery multiplication of Mon_Pro(p, R^2, M) is computed in order to get final output. R^2 is greater than M so a remainder can be obtained from R^2 (mod M) first, and then it can be used.

4.4.1 Hardware implementation of Montgomery Modular Multiplication

In order to compute $A * B$ (mod M) the algorithm 4.2 have two different design solutions for hardware implementation. The inputs to both of the designs are A, B and M and the output is P as shown in figure 4.1,

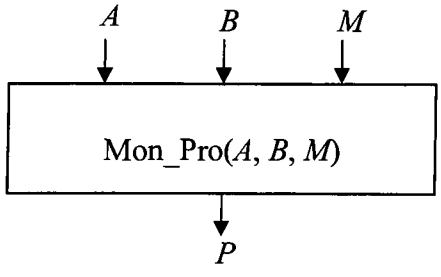


Figure 4.1: Top level block diagram of Montgomery modular multiplier

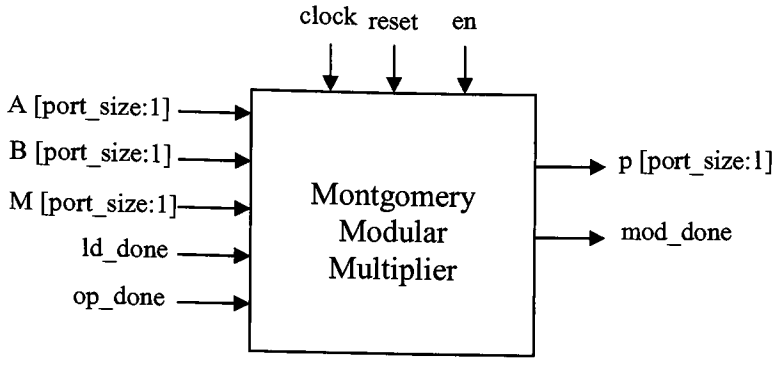


Figure 4.2: Port level block diagram of Montgomery modular multiplier.

Figure 4.2 shows the input and output ports of Montgomery modular multiplier. It has three data input ports and one data output port. Other ports are *ld_done*, *op_done* are used for controlling the data flow, *mod_done* is used as a flag for the completion of the operation and *en* port for enabling and disabling the design.

4.4.1.1 Design with two adders

The first design uses two adders, with two bit multipliers, five registers and a control block as shown in figure 4.3. The first adder is of n+2 bit size, where n= w+2. The second adder is of n+3 bit size. The inputs to the design are A, B and M of w bit size. These inputs are stored in the internal registers by data shift operation. They take small packets of data and serial load them until the whole data is transferred. As the loop in Mon_Pro(A, B, M) runs for n+1 times; the registers storing these values needed to be of the size of n+1. Also A is shifted left up 1 bit before starting the multiplication and reduction operation. An intermediate register S of n+2 bit size is used to keep the intermediate results synchronized. In the algorithm, consider the line 1.3.2,

$$Si = \frac{S_{i-1} + q_i M + b_i A}{2} \tag{4.17}$$

q_i is taken as the LSB of S, and is used as a bit multiplicand in $q_i M$ multiplier. The register storing B is a parallel in serial out shift register, which generates serial bits as b_i for $b_i A$ multiplier. An n+2 bit adder is then used to add $q_i M$ and $b_i A$ multiplication results. The additional 1 bit in this adder is required to save the last carry out. Further the output of n+2 bit adder is added with S_{i-1} to produce S_i. S_i is the output of n+3 bit adder

with the LSB of this adder is discarded in every clock cycle. Again, the additional bit in this adder is required to save the carry out if produced from the addition.

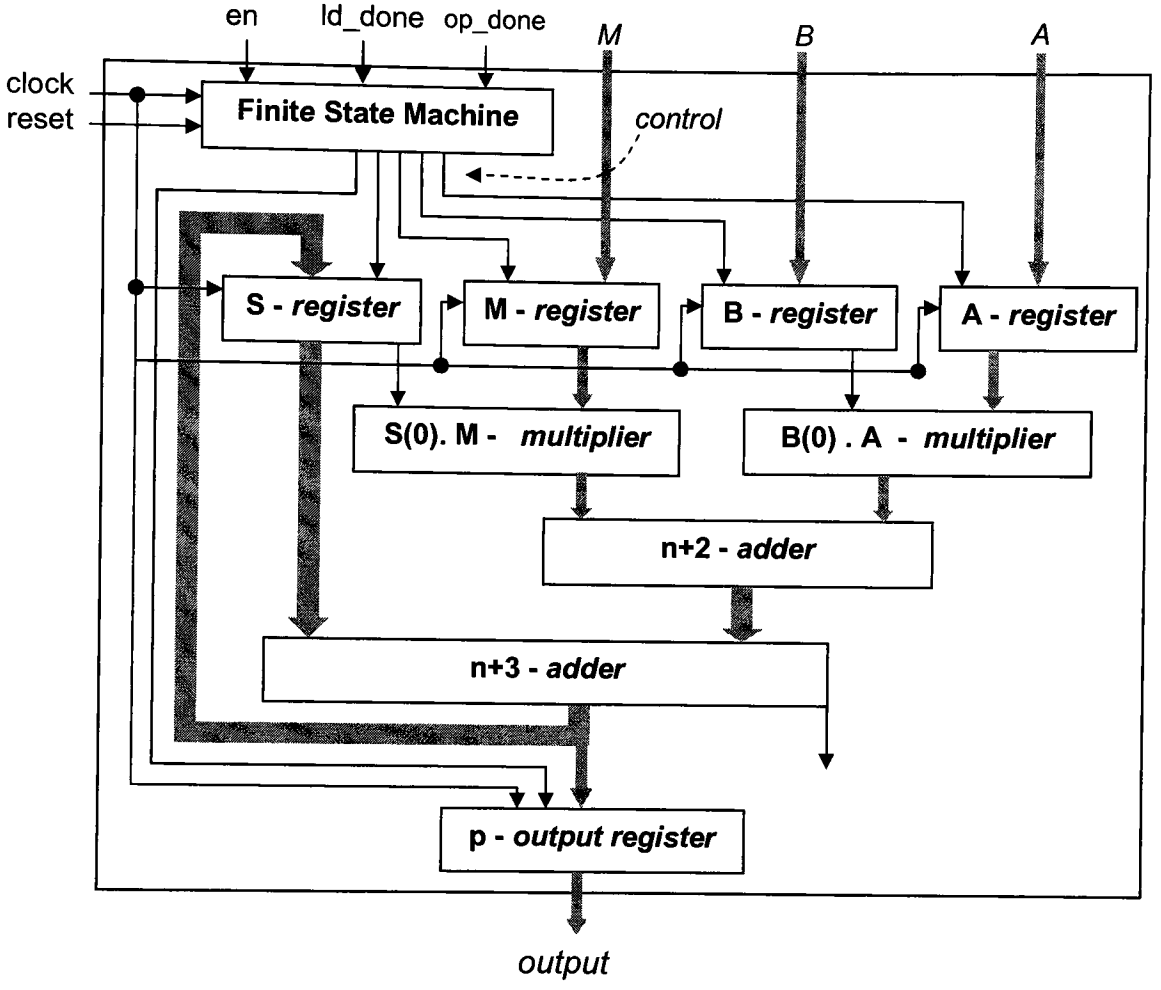


Figure 4.3: Block level diagram of Montgomery Modular Multiplier using two adders and two multipliers.

The iterations are controlled by an n+1 bit counter used in the state machine. The state machine controls the five registers. The output register generates output after n+1 number of counts. Figure 4.2 shows the block diagram of a general Montgomery modular multiplier. Figure 4.3 given next shows the detail about the data flow architecture as connected to the FSM.

The two adders in the design shown in figure 4.3 add carry propagation delays in every iteration. n+2 bit adder can be eliminated by using a multiplexer, which will be

shown in the second design for Mon_Pro(A, B, M) algorithm. Figure 4.4 shows the finite state machine used in this design. It has four states,

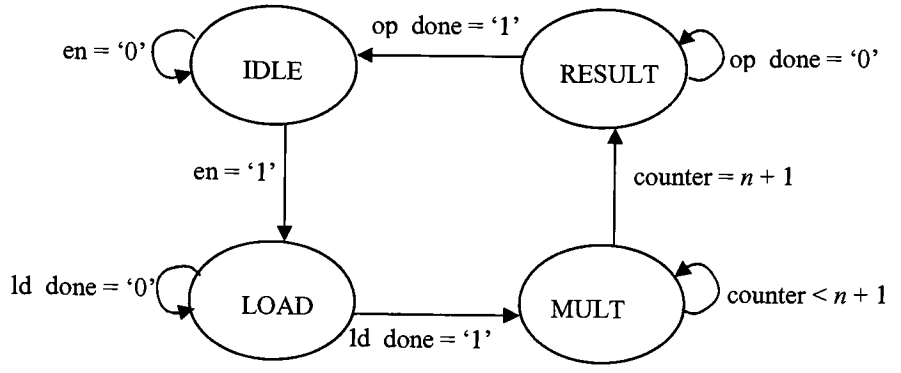


Figure 4.4: Finite state machine for Montgomery multiplier with two adders design.

IDLE:

In this state the FSM remains IDLE, until *en* = '0', the enable signal of the design. Once *en* = '1', FSM clears all the registers, and makes transition to LOAD state.

LOAD:

In LOAD state, the input registers are loaded with the data coming through input ports. The load operation is done serially for small data sets based upon the size of input ports. This makes the design more scalable for an FPGA's available I/O ports. FSM remains in this state until *ld_done* = '0' and keeps loading the registers. Once *ld_done* = '1', it makes a transition to MULT state. During transition, register A is shifted left by one bit, which is equivalent of multiplying by 2.

MULT:

In this state the Montgomery multiplication operation is performed. A counter is used to keep track of all the intermediate additions to be done. It remains in this state, until the *counter* < n+1 and then makes a transition to RESULT state after *counter* = n + 1. During this transition it generates the final output.

RESULT:

In this state, the result obtained in MULT state is transferred out through *output* out port. This is a shift register, which shifts the data out in small data sets. FSM remains in this state until *op_done* = '0'. Once *op_done* = '1', it makes a transition to IDLE state.

4.4.1.1.1 Simulation results

The simulations for design in figure 4.3 are done at RTL and at post-synthesis levels. The RTL simulation does not need any timing constraint, so a minimum of 2 ns clock cycle is used. For post synthesis simulations, the timing constraint is important, and this varies with the size of design. For 4-bit design, the minimum clock period available is 7 ns which is obtained after synthesizing the design which will be discussed in the next section of synthesis results. The timing constraint applies, because of actual wire and component delays caused during the design. The stages of simulations covered in post-synthesis level are post-translate and post-place-route. In order to simulate a 4-bit Montgomery multiplier design in figure 4.3, the date input and the output achieved is as follows,

Input:

$A = (1001)_2 = (9)_{10}$

$B = (1001)_2 = (9)_{10}$

$M = (1011)_2 = (11)_{10}$

Output:

$p = (9)_{10}$

As $p \equiv (A*B)\ R^{-1} \bmod M$, the value of R^{-1} can be calculated using an unsigned version of extended Euclidean algorithm, because a signed version can also returns a negative number and Montgomery modular reduction algorithms only works for unsigned numbers. In order to calculate R^{-1} consider the number of bits in M, which is $w = 4$. Consider $n = w + 2$, so $n = 6$ then for $R = 2^n$, R = 64. Using extended Euclidean algorithm which takes R and M as inputs, the output comes in the form of, $RR^{-1} + MM^{-1} = E$

For relative prime numbers E is always 1. For R = 64 and M = 11, the output from extended Euclidean algorithm achieved was, (5 * 64) – (29 * 11) = 1. The description of calculating R^{-1} here is for verification purpose only. R^{-1} is automatically included as part of algorithm. So using R^{-1} = 5, the output $p \equiv 9 * 9 * 5 \pmod{11} = 9$.

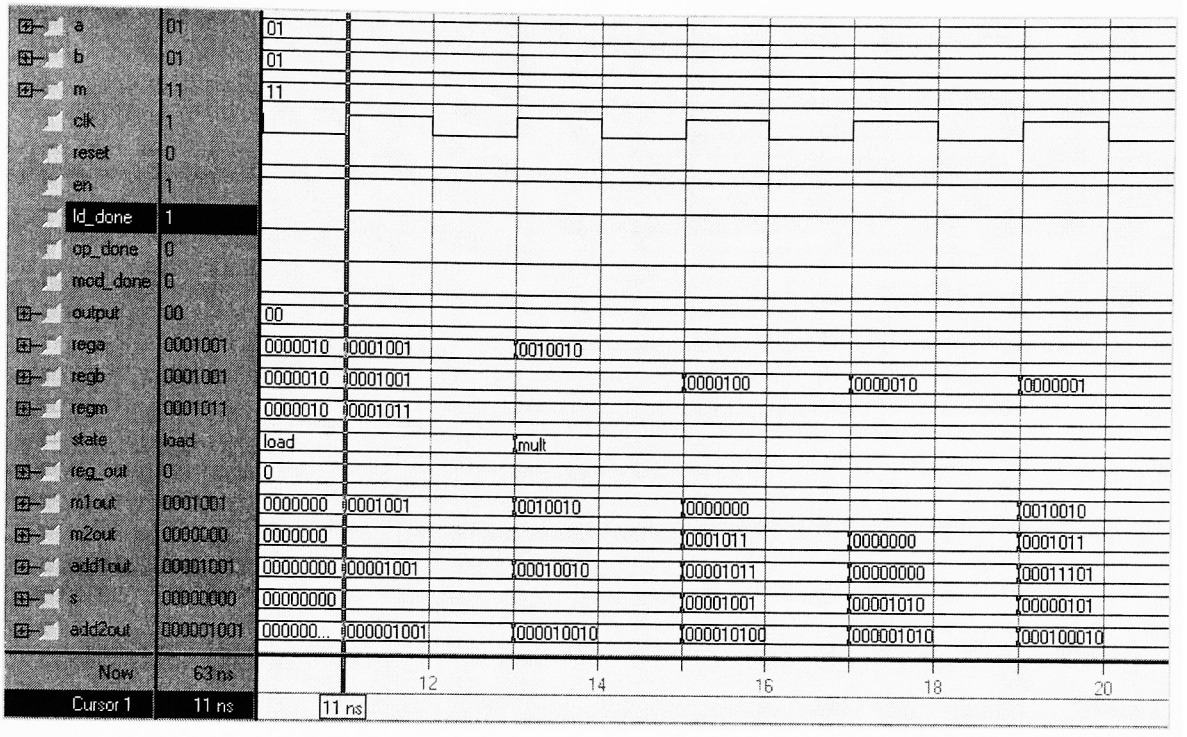


Figure 4.5: Waveform simulations for 4-bit Montgomery modular multiplier design with two adders. Design enable and data load view.

In figure 4.5 of wave form simulations, the *ld_done* signal goes high at 11 ns when the internal registers are loaded with the following input values.

rega = 9

regb = 9

regm = 11

At this moment the FSM is in LOAD state.

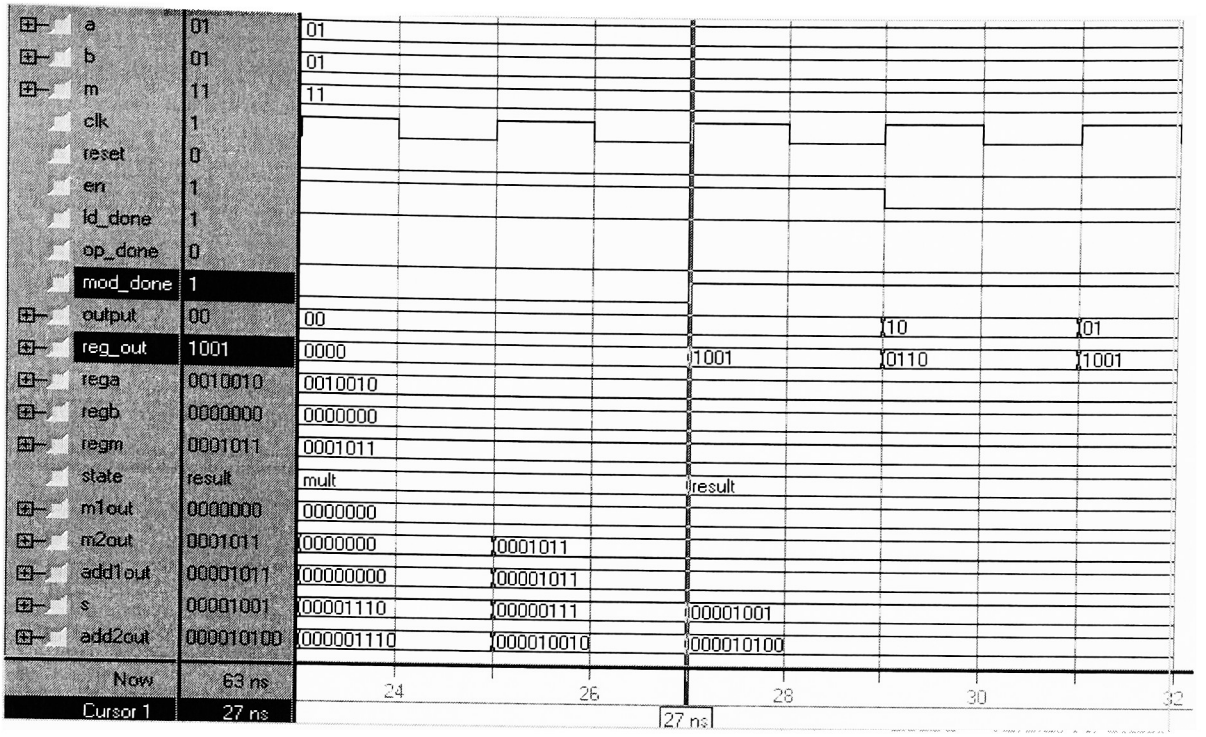


Figure 4.6: Waveform simulations for 4-bit Montgomery modular multiplier using design with two adders. *output* = ABR^{-1} (mod M).

Figure 4.6 shows the *output* = ABR^{-1} (mod M) = 9 is generated and the *mod_done* signal is asserted high. Then the generated *reg_out* = 9 is again fed into the modular multiplier through the stimulus along with R^2 (mod M).

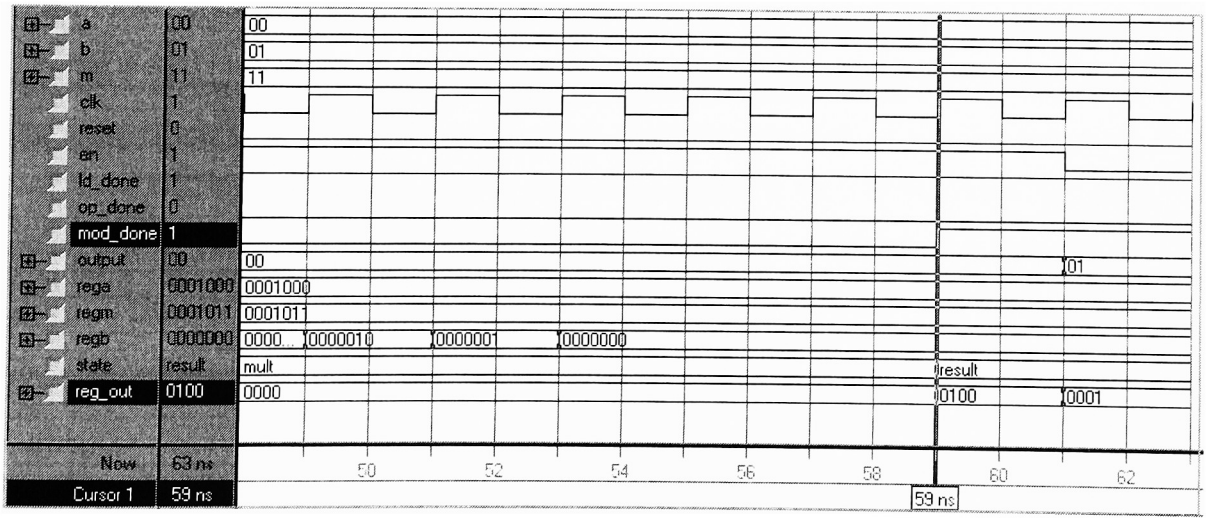


Figure 4.7: Waveform simulations for 4-bit Montgomery modular multiplier using design with two adders. *output* = $AB(R^2)$(mod M).

The waveform in figure 4.7 shows the final result obtained after second Montgomery modular multiplication of the input values,

rega = 9

regb = 4

regm = 11

The output obtained is *reg_out* = 4.

4.4.1.1.2 Synthesis results

The targeted FPGA to synthesize and simulate the design was Xilinx Vertex2p – xc2vp100-5ff1696. Total number of slices available was 44096. The speed grade was selected to be -5 with optimization effort level set to normal for synthesis. The table 4.1 gives the results taken after synthesizing this design in figure 4.3 for 4, 32, 64, 128 and 1024 bits.

Size of Design (bits)	Port Size	No. of Slices	Clock Period (ns)
4	2	76	6.8
32	4	239	8.3
64	8	436	9.5
128	16	824	12.3
1024	32	6401	51.8

Table 4.1: Comparison of FPGA resources used with minimum clock.

4.4.1.2 Design with two adders and a multiplexer

This design uses almost same components as of the multiplier shown in figure 4.3 with some changes and an addition of one major multiplexer. In the modular multiplier design of figure 4.3, the two bit multipliers uses $S(0)$ and $B(0)$ as operands to be multiplied with M and A respectively. This in turn is fed into the n+1 bit adder, which further generates output for the n+2 bit adder. The n+1 bit adder leads to the four possible outputs.

- 0, → when `S(0) = '0' and B(0) = '0'`
- A, → when `S(0) = '0' and B(0) = '1'`
- M, → when `S(0) = '1' and B(0) = '0'`
- M+A, → when `S(0) = '1' and B(0) = '1'`

Because of these four possibilities, the combination of $M.S$ (0), $A.B$ (0) multipliers and the n+1 adder can be replaced by a 4:1 multiplexer with one input connected to an adder M+A. This M+A input adder has an advantage over the old n+1 adder in terms of its carry propagation delay can be considered only at the beginning of the operation, while n+1 will add carry propagation delay for whole of the modular multiplication operation. The reason that M+A will add the delay only at the beginning is because both M and A registers remains constant with their outputs till the completion of the design cycles to generate output. The following figure 4.8 represents the block diagram of this multiplier. In this diagram, the M+A adder also generates the last carry out of M and A addition, so this adds one extra bit to the input size of the multiplier. Because of this, the size of the inputs of multiplexer are made equal to the size of register S, i.e. n+1 bit. When the select lines of the multiplexer have bits "00", the output of the multiplexer is 0, because the input is connected to ground as shown in figure 4.8. It uses the same algorithm as used for figure 4.3. Another change as compared to the design in figure 4.3 is the input registers. This time, the registers are designed to load data in serial small data sets. Thus an extra feature has been added which requires some additional clock cycles to load the data in and also to transfer the final result out. The finite state machine in this design is

same as for the previous design shown in figure 4.4 except a little modification in the LOAD state, where the addition of A and M is performed once *ld_done* = '1'.

Block Diagram:

Consider figure 4.8 for the operational block diagram of the Montgomery modular multiplier. The outputs of the registers *A* and *M* are parallel in parallel out registers, while *B* is a parallel in and right serial out shift register which provides the LSB select signal for the multiplexer. These registers are controlled by the finite state machine for clearing, loading, shifting and holding of the data. The following line of VHDL code shows how the initial data is loaded through input ports.

```
regA  <= regA((k+2) - port_size) downto 0) & A;
regM  <= regM((k+2) - port_size) downto 0) & M;
regB  <= regB((k+2) - port_size) downto 0) & B;
```

Register S is a parallel-in-parallel-out shift register, where its LSB is used as a MSB select signal for the multiplexer. The output register is also controlled by the finite state machine, which generates output after $n + 1$ number of clock cycles. The structure of the output shift register can be understood from the following VHDL line of code.

```
output <= reg_out(k-1 downto (k - port_size));
reg_out  <=  reg_out((k  -  port_size  -1)  downto  0)  &
reg_out(k-1 downto (k - port_size));
```

In this line of code, *reg_out* register is used to store final result produced during MULT state of FSM.

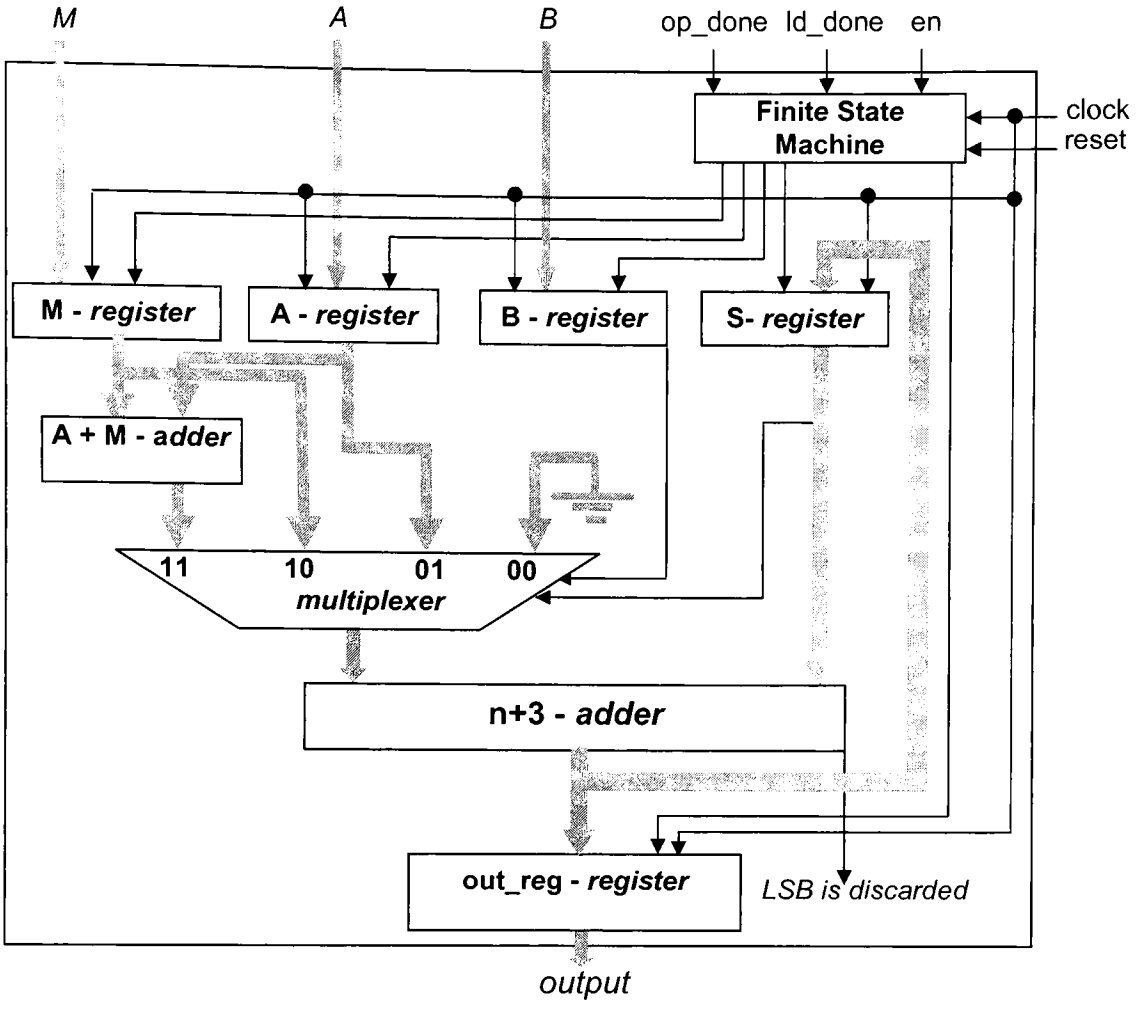


Figure 4.8: Block diagram of Montgomery modular multiplier using multiplexer, and two adders.

4.4.1.2.1 Simulation results

The design in figure 4.8 was simulated at RTL and post-synthesis levels for 4 and 1024 bits of size. The data inputs used in the case of 4 bit design are same as used in the design shown in 4.4.1.1. Consider the simulation results for 4 bit design

- **4 bit Design:**

  Data Input:

  $A = (1001)_2 = (9)_{10}$, $B = (1001)_2 = (9)_{10}$,

  $M = (1011)_2 = (11)_{10}$

Data Output:

Output = p = $(9)_{10}$

The waveform simulations for this design are same as for the design in 4.4.1.1. Similarly the test bench of this design was the same as used in 4.4.1.1. It is also capable of calculating first, $p \equiv (A*B)\ R^{-1} \bmod M$ and then $P \equiv pR^2 \bmod M$.

1024 bit Design:

1024 bit design was simulated to give a performance comparison of Montgomery modular multiplication with an ordinary software based modular multiplication. When compared with ordinary modular multiplication, Montgomery modular multiplication first converts the input multiplicands A & B into Montgomery based numbers. Consider algorithm 4.2, which requires n+1 number of clock cycles to produce the output p. Where $p = A\ B\ R^{-1} \pmod M$. The extra factor of R^{-1} is automatically included due to the algorithm itself. Thus in order to get the final output P of $AB \pmod M$ form, $P = pR \pmod M$ is computed. This requires another Montgomery modular multiplication of $n + 1$ clock cycles for which the inputs will be p, $R^2 \pmod M$, and M. The extra effort of $n + 1$ clock cycles slows down the process of getting final output. Thus a total of $2(n+1)$ clock cycles will be needed. That is why; standalone application of Montgomery modular multiplication is not preferred. The simulation results in table 4.2 provide the speed comparison of hardware and software implementation of Modular multiplication. Total number of clock cycles required for the hardware design was 2191.

- Simulation, HW & SW	**Time**
Hardware Simulation using 62 ns clock period	62* 2191 = 135842 ns = 0.00013 sec
Software Implementation simulation	0.000001 sec

Table 4.2: Comparison of simulation time to complete the modular multiplication operation in hardware and software.

The software implementation simulation time given in table 4.2 is very less as compared to the hardware. This proves that, Montgomery modular multiplication is not suitable for hardware implementation if it is targeted as standalone application. On the other hand, it can be used for implementing modular exponentiation in hardware, which becomes very fast as compared to the software implementation of modular exponentiation. It has been mentioned in the next topic of Montgomery modular exponentiation.

4.4.1.2.2 Synthesis results

The following table gives the results taken after synthesizing this design for 4, 32, 64, and 1024 bits. The FPGA used in the case was Xilinx Vertex2p – xc2vp100-5ff1696.

Size of Design (bits)	Port Size	No. of Slices	Clock Freq (ns)
4	2	83	6.3
32	4	298	7.4
64	8	510	8.4
128	16	969	10.7
1024	32	8050	50.1

Table 4.3: Comparison of FPGA resources used for Montgomery modular multiplier with multiplexer.

A comparison between table 4.1 and 4.3 gives a performance and device utilization improvement in the second design in figure 4.5. The following chart 4.1 gives a comparison for clock speed requirements for the two designs, one without multiplexer (figure 4.3), and one with multiplexer (figure 4.8).

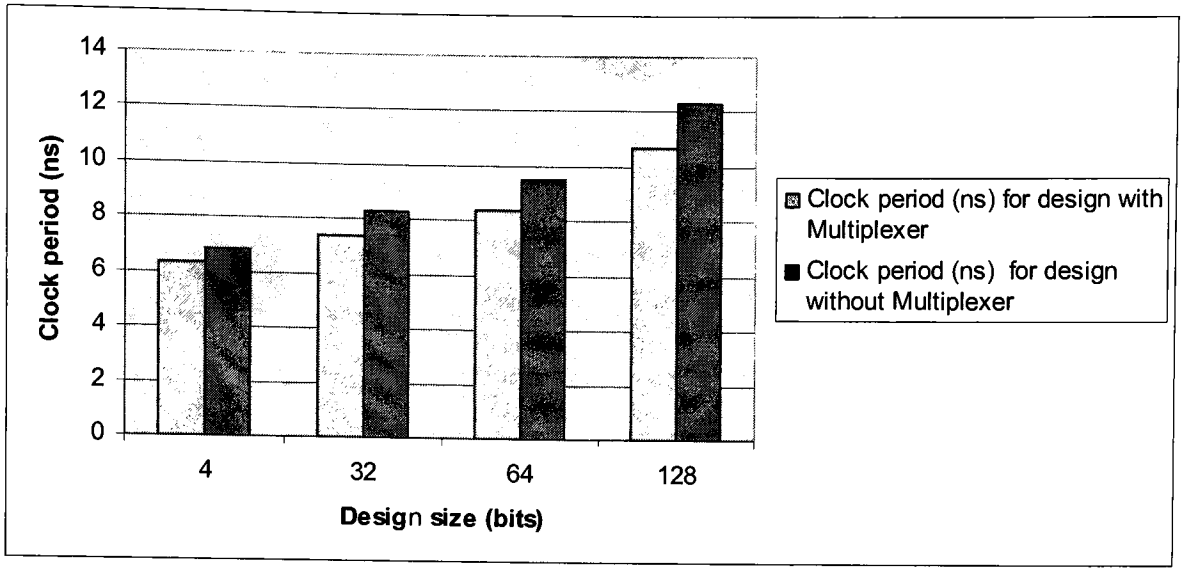


Chart 4.1: Comparison between two designs of Montgomery modular multipliers for minimum clock period required. The sizes of design are, 4, 32, 64 and 128 bits.

In chart 4.1, the design which used multiplexer in figure 4.8 in order to select the inputs for n+3 bit adder gives more speed up as compared to the design in figure 4.3. The design without multiplexer required an adder of n+1 bits to be connected in the data propagation path for every clock cycle, so this added carry propagation delay during every clock cycle. Instead in the design using multiplexer, the use of one n+1 bit adder has been shown at the input of the multiplexer. This adder avoids the carry propagation delay in every clock cycle. An algorithm for modular exponent, which will be discussed later, requires the use of two multiplier units to be used repeatedly to perform the modular exponentiation operation. Chart 4.2 gives a comparison of the number of slices used by each design.

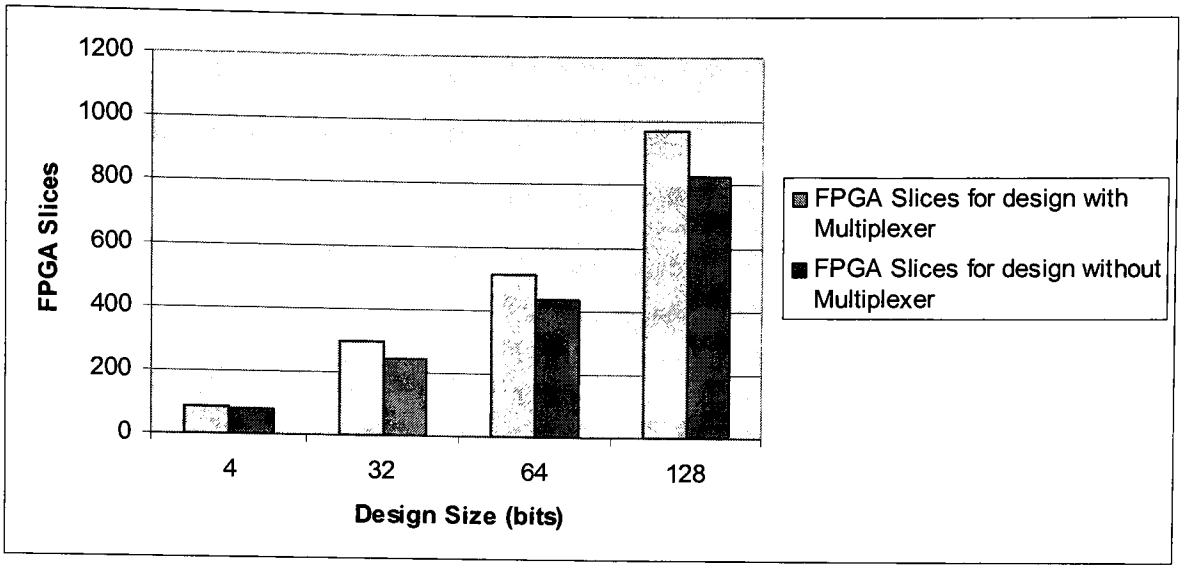


Chart 4.2: Comparison between two designs of Montgomery modular multipliers for number of slices used. The sizes of design chosen are, 4, 32, 64 and 128 bits.

Modular multiplier with multiplexer consumes a little more area as compared to the design without multiplexer. This added area is due to the multiplexer. Thus the second design in figure 4.8 does not eliminate the second adder, but instead it eliminates the effect of carry propagation delay of this adder in every clock cycle. The design goal in this research is to obtain maximum speed, thus the design in figure 4.8 will be referred in constructing the Montgomery modular exponentiation block in the next sections.

4.5 Montgomery Modular Exponentiation

Modular exponentiation has been already discussed in chapter 3, section 2.1.6. There are two basic types of exponentiation algorithms available, right-to-left and left-to-right exponentiation algorithm. Montgomery modular exponentiation is based upon these two algorithms, which uses the basic concept of square and multiply. Consider first left-to-right Montgomery modular exponentiation algorithm.

4.5.1 Left-to-Right Montgomery modular exponentiation algorithm

In order to compute $Z = P^E \pmod{M}$, the inputs to this algorithm are P, E, M, and C. P is the base, E is the exponent, M is modulo, and C is a constant. The representation

of P, E, M and C is in binary or radix 2. Constant C is equal to $2^{2n} \pmod{M}$. This constant must be pre-computed. This exponentiation algorithm repeatedly uses the Montgomery modular multiplier. As generally, in the case of Montgomery modular reduction, whenever a number A is reduced by a modulo M, it is converted to the M residue according to the following equation,

$$\mathcal{A} = A\,R \bmod M \tag{4.18}$$

Thus, for Montgomery reduction, R^2 is required in order to get $\mathcal{A}$ as M residue. For this purpose, $C = R^2$ is fed as input the Montgomery modular exponentiation. Now this gives the exponent function as the Montgomery modular multiplication,

$$\begin{aligned} \text{Monpro}(A, R^2, M) &\Rightarrow A \,.\, R^{-1} R^2 \pmod{M} \\ &\Rightarrow A.\, R \pmod{M} \\ &\Rightarrow \mathcal{A} \end{aligned}$$

R^{-1} is the inverse of R automatically computed during the algorithm computation while performing the Montgomery modular multiplication.

Now considering the algorithm, this constant C needs to be pre-computed. This can be done behaviorally using the software approach outside of the physical design implementation. The other variables used with in the algorithm are, H as an intermediate variable used for the multiplication portion of square and multiply operation with in the loop. w is the number of bits in M.

Algorithm 4.3: MonExpo (P, E, M, C)

1. $H \leftarrow 0$
2. P = Monpro (C, P, M)
3. H = Monpro (C, 1, M)
4. for i = w-1 down to 0 loop

H = Monpro (H, H, M) ....(*Square*)

if ($E(i)$ = '1') then, H = Monpro (H, P, M) ....(*Multiply*)

2. H = Monpro (1, H, M)
3. Return (H)
4. end MonExpo.

This algorithm requires two Montgomery modular multipliers for hardware implementation. Lines 2 and 3 in the algorithm can be computed in parallel, because there is no data dependency exists which can cause a conflict for making the parallel architecture at this point. Further in the loop, the lines 4.1 and 4.2 have a data dependency, which is read after write (RAW) with in the same iteration. Thus here only one multiplier can be used to perform both of the operations of H = Monpro (H, H, M) and H = Monpro (H, P, M) successively. This reduces the area requirement, but slows down the exponentiation operation. At line 5, the computation of H = Monpro (1, H, M) is done in order to remove the effect of R^{-1}, this again requires a multiplier, but as this step comes in last, thus anyone of the old multipliers can be re-used. The computational effort of left-to-right binary exponentiation algorithm can be calculated by first considering the computation effort of each Montgomery modular multiplier, which is n+1. Total computation effort of the above algorithm is thus, $2(n+1)(w+1)$. For example if w = 512 bits, then the computation effort of the left-to-right algorithm will be 528390 clock cycles at minimum. This calculation is not exact, when the algorithm is implemented in hardware some additional clock cycles are needed for loading, shifting, and updating the registers.

Left-to-right binary modular exponentiation algorithm requires less number of hardware resources, but it is practically slow, when compared to right-to-left binary modular exponentiation algorithm.

4.5.2 Right-to-Left Montgomery modular exponentiation algorithm

Similar to left-to-right algorithm, this algorithm also requires the constant $C = 2^{2n} \pmod{M}$ to be fed into the algorithm. Besides the inputs P, E, M, and C, the other variables used in the algorithms are H as an intermediate variable; w is the number of bits in M.

Algorithm 4.4: MonExpo (P, E, M, C)

1. H ← 0
2. P = Monpro (C, P, M)
3. H = Monpro (C, 1, M)

4. for $i = 0$ to w-1 loop
    4.1 if($E(i) = 1$)then
        4.1.1 H = Monpro(H, P, M) ....(*Multiply*)
    4.2 P = Monpro (P, P, M) ....(*Square*)
5. H = Monpro (1, H, M)
6. Return (H).
7. end MonExpo.

This algorithm gives advantage over the previous left-to-right algorithm in terms of speed. This also requires two multipliers like needed in the left-to-right modular exponentiation algorithm. Lines 2 and 3 can be executed in parallel using two hardware Montgomery modular multipliers. Once the computation is done, these multipliers are further used in computing lines 4.1(*Multiply*) and 4.2 (*Square*). Thus here the advantage of more speed is achieved, because both of the lines in the algorithm have no such data dependencies, which can conflict with the parallelism at this point. Finally, at line 5 anyone of the two multipliers can be used again in order to remove the effect of the R^{-1}.

The computational effort of right-to-left Montgomery modular exponentiation algorithm can be calculated by first considering the computation effort of the modular multiplier, which is n+1. Due to parallelism in the loop, the computational effort reduces to (n+1)(w+2). For example, if w= 512 bits, then this algorithm will take 264710 clock cycles at minimum to compute the final result, while the algorithm 4.3 requires 263680 more clock cycles as compare to the right-to-left algorithm, which is almost twice more effort.

After the analysis of speed, the algorithm, right-to-left has been accepted to be implemented for this thesis. For cryptography, speed is the major requirement, and presently the use of 2048 bit RSA and DSA public key cryptosystems require very fast hardware architectures. So the major concern behind the selection of the algorithms is speed when they are implemented in hardware.

4.5.2.1 Hardware implementation of Right-to-Left Montgomery modular exponentiation algorithm

Selection of a fast modular multiplier is important for the hardware implementation of right-to-left modular exponentiation algorithm. Thus, the design shown in figure 4.8, the multiplexer based design is the target multiplier to be used for algorithm 4.4. Before implementing at register transfer level (RTL), the functional verification of the algorithm was very important. For this, a C++ code was created for verifying the algorithm correctness. Also in VHDL, designing the exponent block needed several steps, as initially, it was implemented behaviorally and then was further refined down to RTL. When implemented behaviorally, the design was not provided with a clock, and so the output was based upon the execution of loop statement in the code. When designing at RTL, clock was added along with input and output load and shift registers, so that the data flow could be done in a pipelined fashion. Consider figure 4.9 for the input and output ports configuration, and figure 4.10 for different functional blocks used in the Montgomery modular exponentiation design.

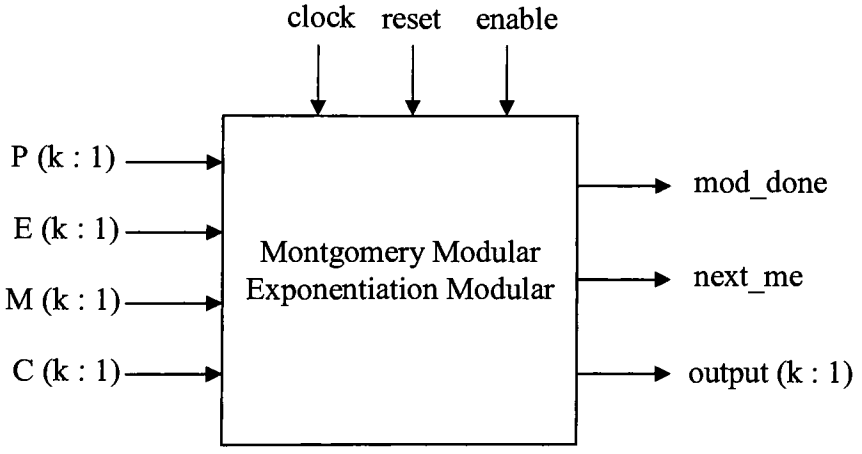


Figure 4.9: Input and output ports of the Montgomery modular exponentiation design shown in algorithm 4.4.

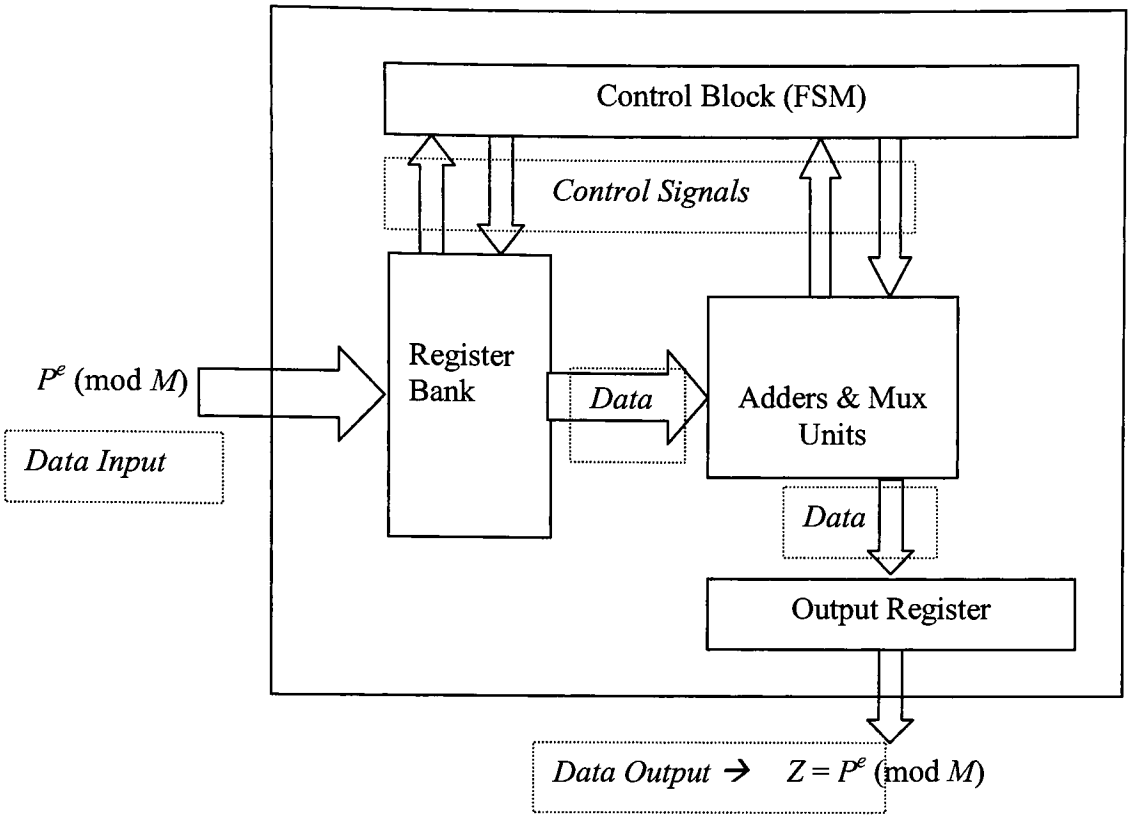


Figure 4.10: Block diagram of right-to-left Montgomery modular exponentiation algorithm

In order to computer $Z = P^e \pmod{M}$, the input registers are loaded with the data (P, E, M, C) as shown in the register bank. Once the data is loaded, the ALU performs the operation of modular multiplication and finally it produces the result Z through the output register. The explanation of the blocks used in figure 4.10 is as follows.

A. Register Bank

Register bank is a set of five different registers. Besides these five registers, there are some additional registers used in designing the Montgomery modular exponentiation, which will be explained later in the detailed data flow diagram. The register bank is responsible for loading the input data from parallel port, updating the subsequent changes upon the completion of modular multiplications as shown in the algorithm 4.4. The size of these shift registers is the size of the design, which is $n+1$ bit. There are two main categories of these registers, which are parallel-in-parallel-out and parallel-in-serial-out. These registers uses different mode signals as generated by the control block and the

sequence of the selection of the mode signal will be explained in the section of Control block. Each of the shift register is explained as follows,

a) Reg_E

This is a parallel-in-serial-out shift register to which the exponent e of $Z = P^e \pmod{M}$ is loaded during the load state of the finite state machine (FSM). The output of *Reg_E* is fed into the FSM itself for the comparison of every incoming LSB (which is *Reg_E*(0)) to be '1' or not. This represents line 4.1 of the algorithm 4.4. There is a 2-bit control signal (*mode*) of this register, which is written in VHDL as follows,

```
case mode is
     when "00" => s <= (others => '0');
     when "11" => s <= data_in;
     when "10" => s <= s;
     when others => s <= '0' & s(j-1 downto 1);
end case;
```

At "00", *Reg_E* clears itself synchronously, at "11" it loads the data in parallel, at "10", it holds the data which is to avoid any changes at the output, and at "01" it performs right shift operation.

b) Reg_M

This is a parallel-in-parallel-out shift register to which the modulus M of $Z = P^e \pmod{M}$ is loaded during the load state of FSM. The output *Reg_M* is fed into the ALU. It uses 2-bit control signal to switch between different modes of operation. In VHDL, the code for the control signal is as follows,

```
case mode is
     when "00" => s <= (others => '0');
     when "11"=> s <= data_in;
     when others => s <= s;
```

```
end case;
```

At "00" *Reg_M* clears itself synchronously, at "11" it loads the data in parallel, at "10" or "01" it holds the data to avoid any changes at its output regardless of changes to be occurred at its inputs

c) Reg_Acc

This is a parallel-in-serial-out shift register, which has a synchronous preset for a value of 1. Also it has two ports for parallel data in, which are required for loading two different intermediate results of the multipliers as shown in the algorithm 4.4. It uses 3-bit control signals to switch between different modes of operations. The VHDL code for these modes of operations is as follows,

```
case mode is
     when "000" => s <= (others => '0');
     when "001" => s <= (0=>'1', others=>'0');
     when "010" => s <= data_in1;
     when "011" => s <= data_in2;
     when others => s <= '0' & s(j-1 downto 1);
end case;
```

At "000" *Reg_Acc* clears itself synchronously, at "001" it is preset to the value of 1, at "010" it loads data_in1, at "011" it loads data_in2, and for all other mode selections it performs the right shift operation for serial out

d) Reg_1

It is a parallel-in-serial-out shift register, which initially loads the base P of $Z = P^e \pmod{M}$. It has 2-bit control signal which is used to switch between different modes of operations. It has two ports for data in, one is used for loading P and one is used for loading the intermediate results generated by the multipliers as shown in the algorithm 4.4. The VHDL code for these control signals as follows,

```
case mode is
     when "00" => s <= (others => '0');
     when "01" => s <= data_in1;
     when "10" => s <= data_in2;
     when others => s <= '0' & s(j-1 downto 1);
end case;
```

At "00" *Reg_*1 clears itself synchronously, at "01" *data_in1* is loaded, at "10" *data_in*2 is loaded, and at "11" it performs the right shift operation for serial out data.

e) Reg_2

It is a parallel-in-parallel-out shift register, which initially loads the constant *C* as the input of the right-to-left algorithm. It has a synchronous preset for the value of 1, and two parallel data in ports. It uses 3-bit control signal to switch between different modes of operations. This register represents the register *A* in figure 4.7. The VHDL code for these control signals as follows,

```
case mode is
     when "000" => s <= (others => '0');
     when "001" => s <= data_in1(j-2 downto 0) & '0';
     when "010" => s <= data_in2(j-2 downto 0) & '0';
     when "100" => s <= (1=>'1', others=>'0');
     when others => s <= s;   --hold
end case;
```

At "000", *Reg_*2 clears itself synchronously. At "001" it loads *data_in*1 in parallel and also it shifts to the left by 1 bit, this is because in the algorithm 4.2 register *A* is multiplied by 2 which is equivalent to be shifting a register to the left by 1 bit. Similarly at "010" it loads *data_in*2 while shifting it to the left by 1 bit. At "100", it presets itself to

the value of 1, and at all other cases it hold the data which has been loaded in order to avoid any changes at the output regardless of changes occur at the input.

B. ADD_MUX block

This block is composed of two identical multiplexers, four adders, and three intermediate parallel-in-parallel-out shift registers.

a) Multiplexers and Adders

The configuration of a multiplexer is 4:1 with n+2 bit data I/O. It is similar to the multiplexer used in the Montgomery modular multiplier in figure 4.8. The adders are also similar adders as in figure 4.8. Two adders are connected at the input of two multiplexers with n+2 bit data out, while the other two adders are connected at the output of both of the multiplexers with n+3 data out.

b) Shift Registers

Among the three shift registers, two are identical named as *Reg_S*. Each *Reg_S* is connected between the input and output of an n+3 bit adder. So this makes a loop back to the n+3 bit adder with *Reg_S* connected in-between. *Reg_S* is required to synchronize the data in the loop path with the data coming from the multiplexers. It has 1 bit control signal as shown in the following VHDL code,

```
case mode is
          when '0' => s <= (others => '0');
          when others=> s <= data_in;
end case;
```

At '0', the *Reg_S* clears itself synchronously, and at '1' it does parallel shift.

The third shift register is a hold register named as *Hold_reg*. It is only connected at the output of one of the *Reg_S*. This register is required for holding the data if the LSB of PISO register *Reg_E* is '0'. When '0', it holds the previous results of *Reg_S*, until the

LSB = '1' of *Reg_E* arrives. It represents the if statement in the algorithm of right-to-left shift register at line 4.1.

A complete connection configuration of a multiplexer, *n*+2 adder, *n*+3 adder, *Reg_S*, and *Hold_reg* is shown below in figure 4.11.

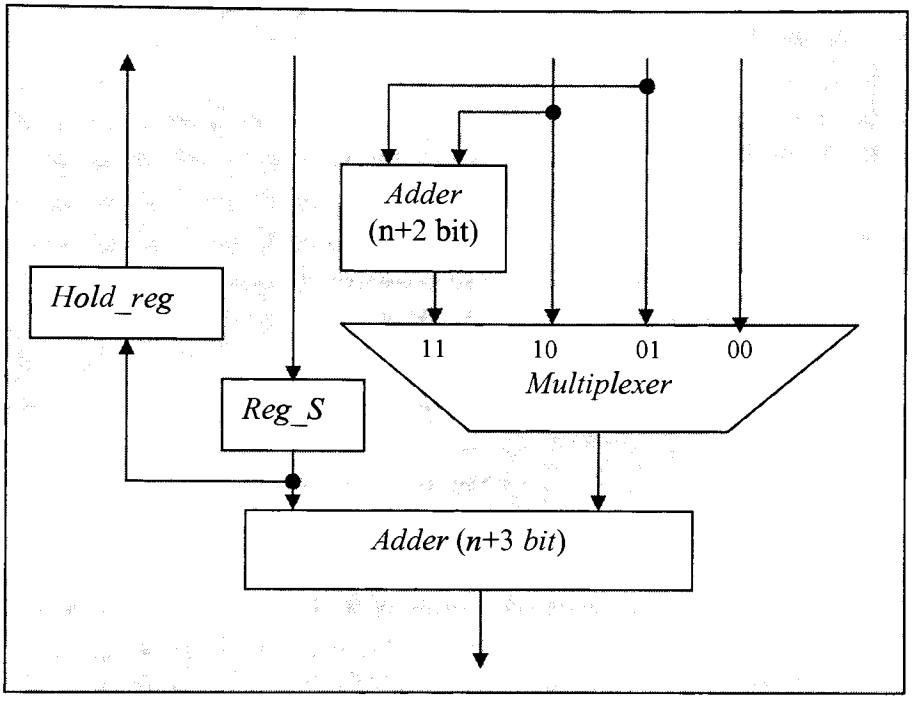


Figure 4.11: Block diagram of one ADD_MUX block used in right-to-left Montgomery modular exponentiation algorithm

Figure 4.11 represents the components used in ADD_MUX block. Two such blocks are required by the modular exponentiation design with an addition of a register named *Hold_reg*. This register is only used in one of the ADD_MUX block which also generates the final output of the modular exponentiation as shown in figure 4.12. Consider figure 4.12 for the complete diagram of Montgomery modular exponentiation based upon the algorithm 4.4.

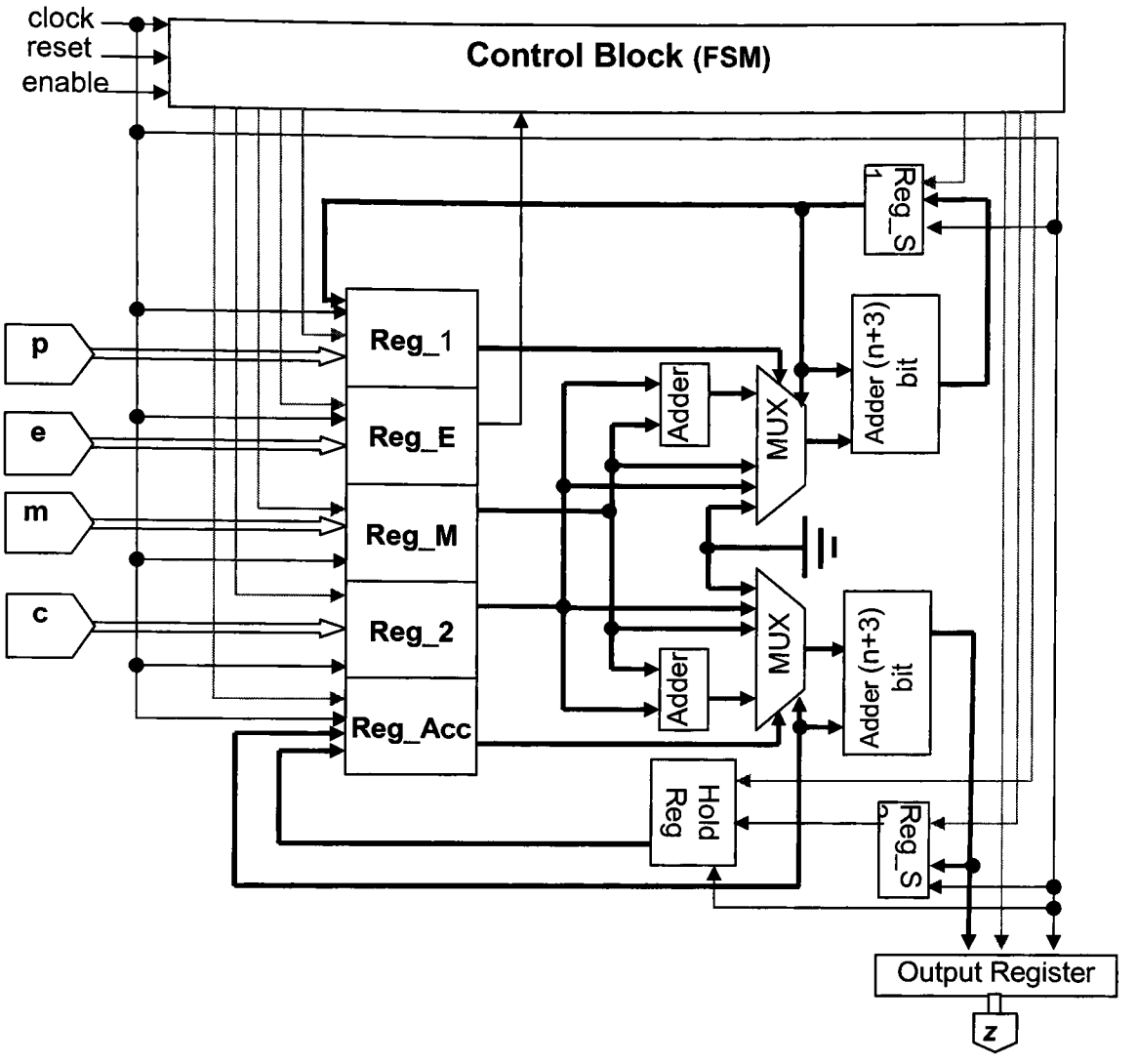


Figure 4.12: Data flow level block diagram of right-to-left Montgomery Modular Exponentiation

C. Control Block (Finite State Machine)

Consider figure 4.13 of the finite state machine used in right-to-left Montgomery modular algorithm. It has five major states and three minor states.

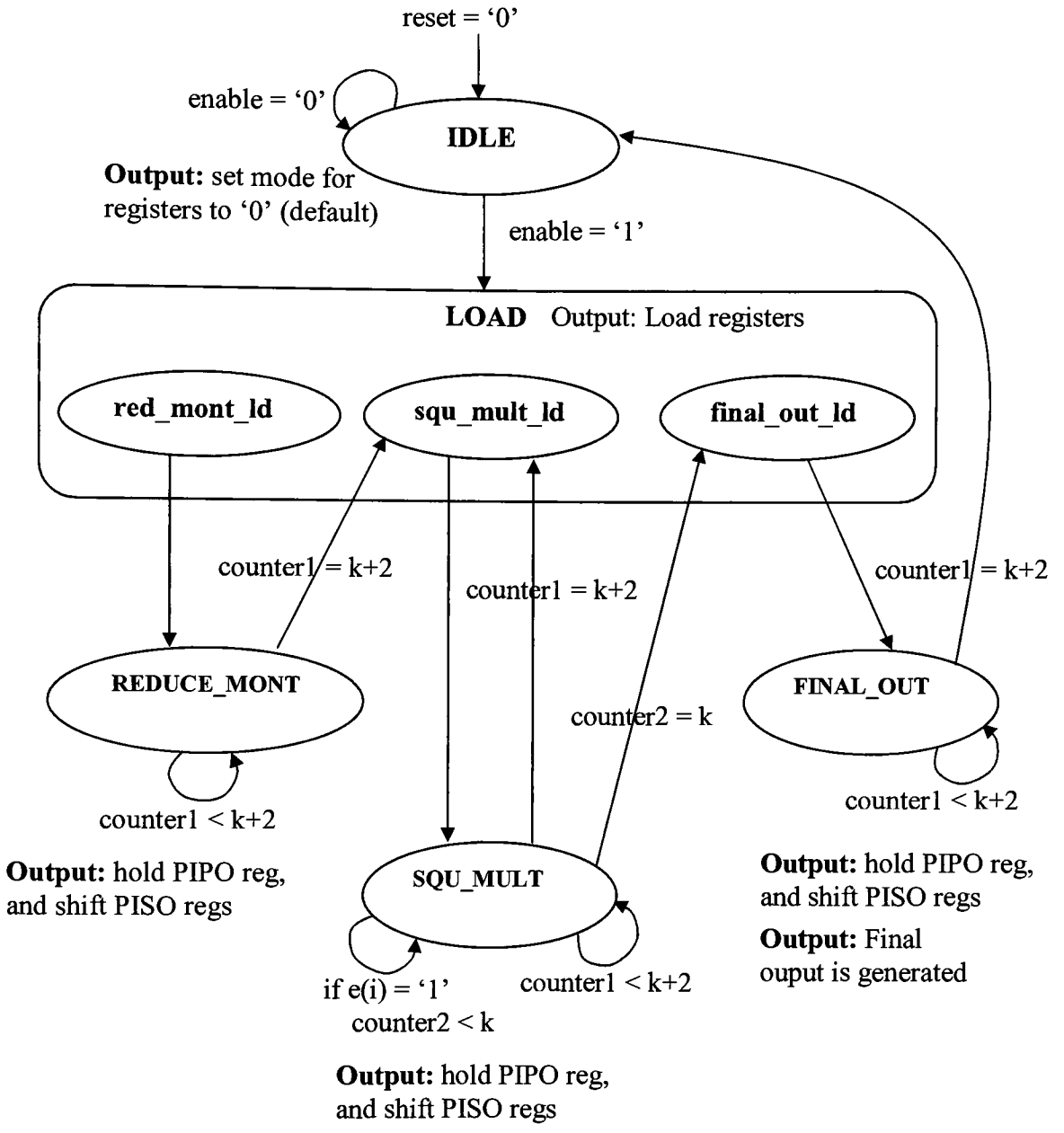


Figure 4.13: Finite State Machine of right-to-left Montgomery Modular Exponentiation Algorithm

The state machine has two internal counters *counter*1, and *counter*2. *counter*1 keeps track of the complition of Montgomery multiplication after n+1 clock cycles. *counter*2 controls the execution of square and multiply phase of Montgomery exponentiation as shown in line 4 of the algorithm 4.4. It runs for w clock cycles.

Before explaining each state consider the table 4.4 for the control signals of all the shift registers in the design.

	Clear	Data_1	Data_2	Preset	Hold	Shift right Serial
Reg_1	00	01	10			11
Reg_E	00	11			10	01
Reg_M	00	11			01/10	
Reg_2	000	001	010	100	All others	
Reg_Acc	000	010	011	001		All others
Reg_S1	0	1				
Reg_S2	0	1				
Reg_Hold	00	11			10/01	
Reg_Out	00	11			10/01	

Table 4.4: Control signals for shift registers generated by finite state machine.

a) IDLE

This is the state in which FSM remains when *reset* = '0' and *enable* = '0'. When the state machine completes its all tasks, it comes to this state. When *enable* goes high it clears all the output registers and then change the state of the FSM to LOAD.

b) LOAD with red_mont_ld

Load state is further composed of three sub-states. These sub-states are used for loading Montgomery multipliers used at various places in the algorithm 4.4. The default sub-state of LOAD state is *red_mont_ld*. In *red_mont_ld*, the registers for P = MonPro(C, P, M) and H = MonPro(C, 1, M) are loaded with values. The following table gives the control signals generated in this sub-state to load the values. After generating these control signals, the FSM changes to state REDUCE_MONT.

Registers	Data_1	Preset
Reg_E	11	
Reg_1	01	
Reg_M	11	
Reg_2	001	
Reg_Acc		001
Reg_hold	11	

Table 4.5: Control signals generated by FSM for registers in design in LOAD with *red_mont_ld* sub-state.

c) REDUCE_MONT

In REDUCE_MONT, P = MonPro(C, P, M) and H = MonPro(C, 1, M) are computed. This name of the state represents the operation of Montgomery reduction of P to a Montgomery based reduced P, and also to generate H which is further used in square and multiply stage of the algorithm. The table for the control signals generated in this state is as follows with the active signals are highlighted in white boxes.

Registers	Data_1	Hold	Shift right Serial
Reg_1	~~01~~		**11**
Reg_M	~~11~~	**01/10**	
Reg_2	~~001~~	**All others**	
Reg_S1	**1**		
Reg_S2	**1**		

Table 4.6: Control signals generated by FSM for registers in REDUCE_MONT state.

In REDUCE_MONT *counter*1 is incremented on every clock cycle till it reaches w+3 count. This completes the Montgomery multiplication operation for P and H as

show on lines 2 and 3 in the right-to-left algorithm. After that FSM changes the state to LOAD with its substate *squ_mult_ld*.

d) LOAD with squ_mult_ld

In squ_mult_ld, the registers for square and multiply stage are loaded. The following table presents the control signals generated for the registers of the design.

Registers	**Clear**	**Data_1**	**Data_2**
Reg_1	~~00~~	~~01~~	**10**
Reg_2	~~000~~	~~001~~	**010**
Reg_Acc	~~000~~	**010**	**011**
Reg_S1	**0**	~~1~~	
Reg_S2	**0**	~~1~~	

Table 4.7: Control signals generated by FSM for registers in LOAD with squ_mult_ld state.

In table 4.7 Reg_Acc has a condition to either set at "010" or "011". The condition is based upon Reg_E serial output. It Reg_E(0) = '0' then Reg_Acc is set at "011" otherwise it is at "010". Both of Reg_S1 and Reg_S2 are cleared in this state for the next Montgomery multiplicaiton operation. After generating the above control signals the FSM changes to the state of SQU_MULT.

e) SQU_MULT

This state performs square and multiply portion of the algorithm 4 .i.e.

if($E(i) = 1$)then, $H = \text{Monpro}(H, P, M)$; ....(*Multiply*)

$P = \text{Monpro}(P, P, M)$; ....(*Square*)

If the LSB of *Reg_E* is '1', then FSM executes $H = \text{Monpro}(H, P, M)$. For this it loads *Reg_Acc* with the H and *Reg_2* with P generated from the previous Montgomery

modular multiplications. For P = Monpro (P, P, M), the FSM loads *Reg_*1 with P and as *Reg_*2 is already loaded with P as well so it remains in hold condition. The following table shows the control signals generated by FSM for the registers used for this portion of algorithm.

Registers	**Clear**	**Data_1**	**Hold**	**Shift right Serial**
Reg_1	~~00~~	~~01~~		**11**
Reg_E	~~00~~	~~11~~	~~10~~	**01**
Reg_2	~~000~~	~~001~~	**All others**	
Reg_Acc	~~000~~	~~010~~		**All others**
Reg_S1	~~0~~	**1**		
Reg_S2	**0**	**1**		
Reg_Hold	~~00~~	**11**	**All others**	

Table 4.8: Control signals generated by FSM for registers in SQU_MULT state.

In table 4.8, the values in white boxes are the modes asserted during this state. Notice *Reg_S*2, which is '0' when *Reg_E*(0) = '0'otherwise *Reg_S*2 = '1'. Also *Reg_Hold* is "01/10" when *Reg_E*(0) = '0', otherwise it is "11" which loading new data. This is required because if the previous to the present LSB of *Reg_E* is '1', then H = Monpro(H, P, M) executes, and so if the present LSB is '0' then *Reg_Hold* holds the previous data H to be used for the next Montgomery modular multiplication and also during this time *Reg_S*2 is set to '0', which is to keep its output set to the value of 0. The condition that applied to *Reg_S2* avoids any unwanted results to be appearing at the input of *Reg_Acc* which can load a wrong value for the coming multiplication.

The control signal for *Reg_E* is also set to "01", which is a right serial shift. This is done, when ever a Montgomery multiplication is completed with in the range of *counter*2. As mentioned earlier, *counter*2 runs for w times. Thus *Reg_E* performs one right serial shift after every w number of clock cycles. *counter*1 increments on every clock cycle until it reaches w+3 then with in SQU_MULT state FSM verifies that if

*counter*2 < *w* or not. If it is less then *w*, then the FSM chages its state from SQU_MULT to LOAD with squ_mult_ld state in order to load new outputs generated from the previous modular multiplications. On the other hand, if *counter*2 completes its *w* iterations, the FSM changes the state to LOAD with final_out_ld state.

f) LOAD with final_out_ld

In this state values for the final Montgomery multiplications are loaded. The table 4.9 shows the control signals, which are generated in this state.

Registers	**Clear**	**Data_1**	**Data_2**	**Preset**
Reg_2	~~000~~	~~001~~	~~010~~	**100**
Reg_Acc	~~000~~	~~010~~	**011**	~~001~~
Reg_S1	**0**	~~1~~		
Reg_S2	~~0~~	**1**		

Table 4.9: Control signals for registers in LOAD with final_out_ld state.

Reg_S2 is synchronously preset to the value of 1 by generating control signal of "100". This represents *H* = Monpro (1, *H*, *M*) at line 5 of right-to-left Montgomery modular exponentiation algorithm. Once the registers are loaded, the FSM does the transition to
FINAL_OUT state.

g) FINAL_OUT

This state performs *H* = Monpro (1, *H*, *M*) operation. The control signals generated in this state are shown in table 4.10.

Registers	Clear	Data_1	Hold	Shift right Serial
Reg_2	000	001	All others	
Reg_Acc	000	010		All others
Reg_S2	0	1		
Reg_Out	00	11	10/01	

Table 4.10: Control signals generated by FSM for registers in FINAL_OUT state.

As it is seen in the table 4.9. *Reg_S*1 is not displayed. It is because only one modular multiplying unit is active now. Also for *Reg_*2, the hold mode is asserted, because the previous value of *H* from the SQU_MULT state is now used in the multiplication. *Reg_out* is active now. Once the *counter*1 = *w*+3, after that FSM sets up the *Reg_out* to load the data from the output of the second *n*+3 bit adder connected in *Reg_S*2 block. Once the data is loaded, then FSM sets *Reg_out* to hold mode for the data to be maintained at output. In this state the *mod_done* signal is also asserted to logic '1' showing that the Montgomery modular exponentiation is completed.

4.5.2.1.1 Simulation of Right-to-Left Montgomery modular exponentiation

The VHDL model was simulated using Modelsim at both RTL and post synthesis levels. For post-synthesis simulation, the sub-levels of simulation covered step by step including post-translate simulation, and post place and route simulation. The time taken for the simulation to be completed was (*n*+1) (*w*+2) clock cycles. The simulated data was 4 bit. The inputs shown are,

P <= $(1011)_2$ -- $(11)_{10}$

E <= $(0111)_2$ -- $(7)_{10}$

M <= $(1101)_2$ -- $(13)_{10}$

C <= $(1100)_2$ -- $(12)_{10}$

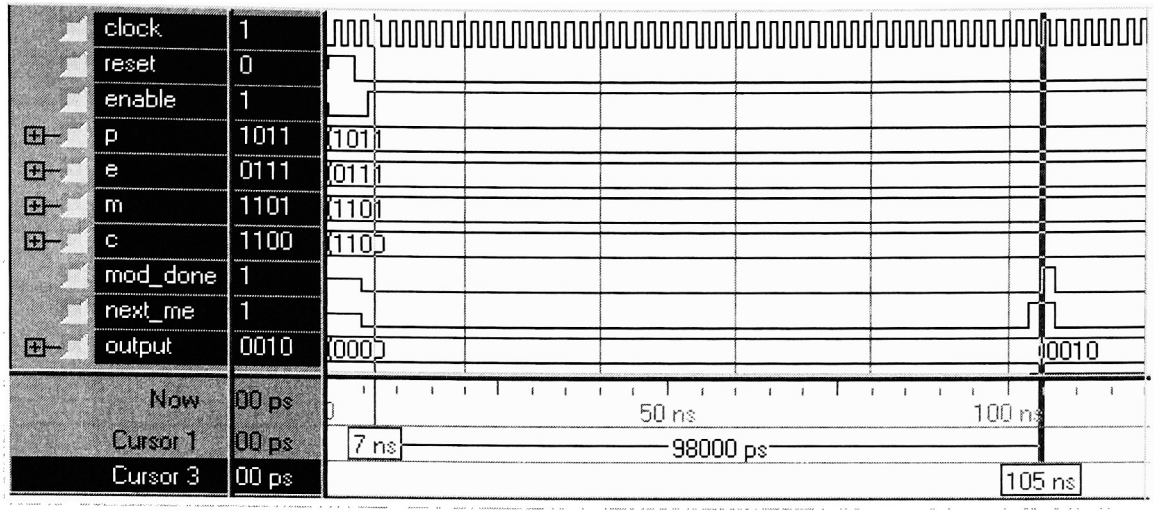


Figure 4.14: Wave form simulations showing the data inputs of p, e, m and c with output generated.

Figure 4.14 displays the output obtained at 105 ns, while the design got enable at 7 ns. Thus a total of 49 clock cycles are used in computing the final output. Considering the algorithm of right-to-left Montgomery modular exponentiation, the total of $(n+1)$ $(w+2)$ clock cycles are required. This is equal to 42 clock cycles. In simulation, the output arrived after 49 clock cycles. A difference of 7 clock cycles has been occurred which is due to the load and shift of the registers used in the register bank, hold register and output register. These 7 clock cycles can be explained by considering the right-to-left algorithm which requires a total of 6 Montgomery modular multiplications when considering the parallel operations. Reg_1 and Reg_2 in the register bank are updated upon the completion of every modular multiplication. Thus 6 clock cycles are consumed by the registers in register bank. An additional 1 clock cycle is needed to generate output, which is consumed by output register.

Figure 4.15 and 4.16 gives the condition of the internal register enable signals with the register values updated. These waveforms do not include the I/O ports. The signal format is displayed in unsigned format.

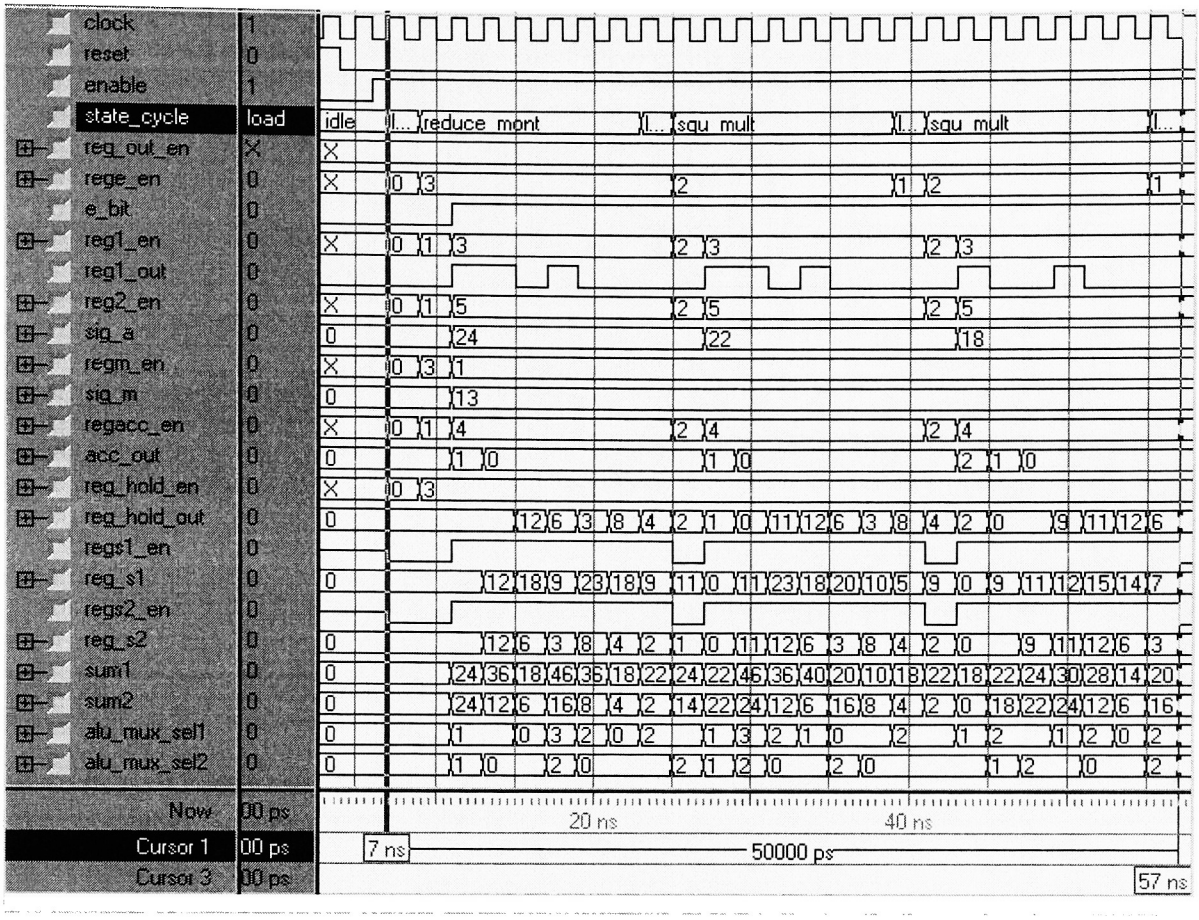


Figure 4.15: Wave form simulations showing the behavior of the registers used in the design. This simulation is the first half of the total simulation.

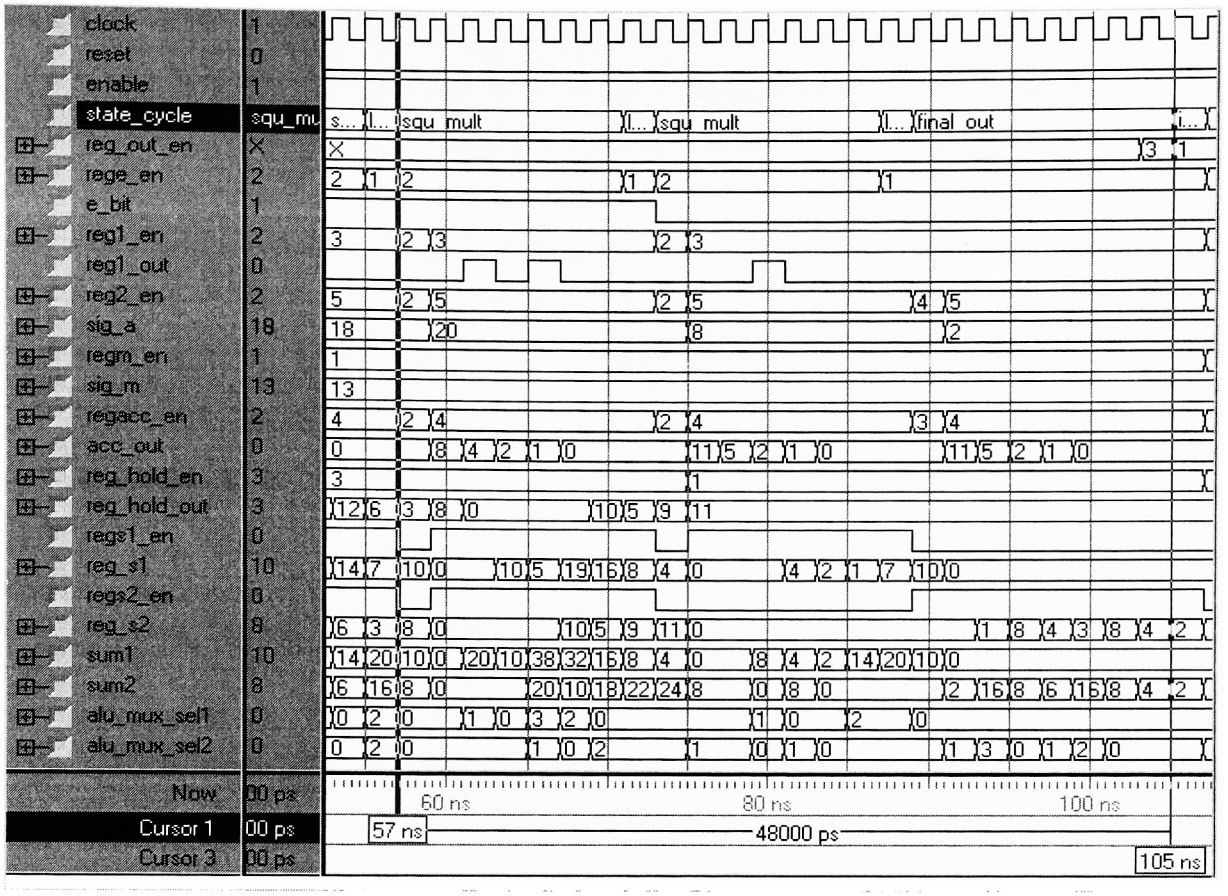


Figure 4.16: Wave form simulations showing the behavior of the registers used in the design. This simulation is the second half of the total simulation.

4.5.2.1.2 Synthesis of Right-to-Left Montgomery modular exponentiation

This design was synthesized for 4, 32 and 64 using Xilinx Vertex2p – xc2vp100-5ff1696 FPGA. There were 33280 slices available on this FPGA. Because of parallel ports used, a synthesis of approximately 124+ bit sized design was not possible. Table 4.11 shows the results taken for various design sizes.

Size of Design (bits)	No. of Slices	Clock Period (ns)
4	141	8
32	462	8.7
64	828	10.1

Table 4.11: Results taken from the synthesis of different sizes of the design of Montgomery modular exponentiation algorithm.

4.5 Summary

The design of Montgomery modular exponentiation explained in figure 4.14 has been used in the implementation of Digital Signature Algorithm (DSA). The design was implemented completely scalable to be incorporated for any size of DSA. In chapter 5, the hardware implementation of DSA has been shown along with its implementation in software. The purpose of implementing DSA in both hardware and software is to show the through put of both of the implementations.

Chapter 5

Hardware and Software implementation of Digital Signature Algorithm using Montgomery Modular methods

Digital Signature Algorithm given in chapter 2 has been implemented in software and hardware. The implementation level in software is completely functional with no timing and signal details. For modular arithmetic, the fast multi-precision software algorithms are used. For hardware implementation, the slow portions of DSA are targeted including the modular exponentiation for modulo p. The hardware portions are then executed in corporation with the software portions for a complete DSA operation.

5.1 Software Implementation of Digital Signature Algorithm

The software implementation of DSA based upon algorithm 2.1 was generic for various sizes of modulus p. The software design was targeted for p = 1024 bits, q = 160 bits and x = 864 bits. Standard C++ does not support data types which are longer than 32 bits. Thus standard C++ did not support the implementation in software. Selection of SystemC as multi-precision C++ language was made as a result of these limitations. In SystemC *sc_bigint* and *sc_biguint* are of arbitrary data size. It means that it supports any size of numbers. SystemC is a C++ library which is designed to support hardware modeling as well as verification. The level of abstraction of modeling can be from functional algorithmic level down to concurrent hardware modeling like hardware description languages. For software implementation, the high level abstraction feature of SystemC was used. The units of software implementation were,

a) main.cpp

It includes the top level detail of the design. In top level detail, the classes are instantiated and their methods are invoked for particular instances.

b) data.h

It is a package class which includes the constant variables and procedures to be used in all other classes as well as in the "main.cpp". The constant variables included are

used to set the sizes of p, q, x, and h (hash) variables. These variables are declared as follows,

```
const int h_width = 160;   --- SHA(m)
const int q_width = 160;   ---q
const int p_width = 1024;  ---p
const int k_width = 864;   ---w
```

"data.h" also has two functions to compute modular exponentiation using square and multiply algorithm.

hash.h

This class implements SHA-1 message digest algorithm. The algorithm is given in [8]. In addition to the implementation of algorithm, this file is also capable of reading the text message from a message file, and then converting the message into numerical format between the range of 0 and 25 with a = 0, and z = 25 all lower case letters with no spaces and no special characters in between them.

c) pre_sign_block.h

This class implements some pre-computations of the signing operation of DSA. These pre-computations include finding primes p and q while selecting an even number x. The operation included here from algorithm 2.1 is,

- $p = qx + 1$

The algorithm used to find prime numbers is Miller Rabin Primality Test.

d) dsa_block.h

This file includes all the major computations of DSA. The computations are,

- $\alpha \equiv g^{(p-1)/q} \pmod{p}$,
- $\beta \equiv \alpha^{a} \pmod{p}$
- $r = (\alpha^{k} \pmod{p}) \pmod{q}$

- $s \equiv k^{-1} (\mathrm{SHA}(m) + ar) \pmod{q}$
- $u_1 \equiv s^{-1} \mathrm{SHA}(m) \pmod{q}$
- $u_2 \equiv s^{-1} r \pmod{q}$
- $v \equiv (\alpha^{u1} \beta^{u2} \pmod{p}) \pmod{q}$
- $v = r$.

dsa_block.h has four functions in it which are *create_sign*(), *verify_sign*(), *find_alpha*(), and *ext_euc*(). *ext_euc*() function based upon extended Euclidean algorithm is used to find the modular multiplicative inverse of secret number *k,* which is required in the algorithm for signature creation operation.

Operation:

In *main.cpp*, digitial signature algorithm starts with first creating the hash by invoking the *hash.h* constructor and calling its function *SHA*1 (). This function in turns calls the *conv_string*(), *pad_msg*() and *hash_funct*() functions and finally returns the message digest. After that in *main.cpp*, the *pre_sign_block.h* is instantiated by invoking its constructor. The function *generate_primes*() is called for this instance of *pre_sign_block.h* in order to generate primes *p*, *q* and the even number *x*. These three numbers are made public which are accssed directly by the *main* () function in *main.cpp* file. Finally, *dsa_block.h* is instantiated in *main.cpp*, which creates the signature, verifies it and generates the output.

The input to this program is a message file, "*msg.txt*" and the output of this program is a data file, "*data_file.txt*". The "*data_file.txt*" is created in *dsa_block.h* in two formats which are binary and decimal.

5.1.1 Results and Analysis of Software Implementation

Table 5.1 shows the output in hexadecimal format for p = 1024 bits, q = 160 bits and x = 864 bits.

Variables	Output values
p	0x0400000000000000000000000000000000000cef8000000000000000000000000000 000000000000000000000000000000000000004000000000000000000000000000 0000000cef80000000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000001
q	0x080000000000000000000000000000000000019df
x	0x0800000000000000000000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000800000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000 000000000
a	0x060000000000000000000000000000000000019df
k	0x070000000000000000000000000000000000019df
β	0x01f5e6add9f32975859c942f09152d96a2673a20ec46216a5148fbf37bee55001903686 2295aa8417bddfd7f254cfa87b1e646e6d88b9e80dc345743 18bc90126f2165b594513de4 02639477e932ef1c9f5d788f9596c738545978ef37cf772aef8452f367c19487a6513d76cf a7e43796810c156b810af3f20c940558ebedd27
α	0x03558df42ac52a3c7775f2000ebe4f9bbe81093a4cc3e8b19ca38d82e77e1a522f98881 adab295f75e1de9e4b48f24503a81b38af99e55ed2db427c24c8f30cf3b3a4640d452220d dc066c0c2155137d38e6c1b4b42d258cbe23e9279eb7350e80a209ab34de7c9779b10d4a 9612313d675622a9d8e475df185e18018f8ff1fa4
r	0x048e275882ab7c8a22efa4e19bc298fa4face7de8
s	0x0195e82962bc98918e8ac48aafbc981574704f5ae
SHA(m)	0x0c3d2e1f09032547618badcfe427656d1a26e04b8

Table 5.1: Values generated at the output of the software implementation of Digital Signature Algorithm.

The following portion of the algorithm was subjected to the timing analysis for a comparison with hardware speed performance.

Signature operation by Alice:

$\alpha \equiv g^{(p-1)/q} \pmod p$, then $\alpha^q \equiv 1 \pmod p$ must be satisfied.

$\beta \equiv \alpha^a \pmod p$

$r \equiv \alpha^k \pmod p$

Verification operation by Bob:

$$v \equiv \alpha^{u1} \beta^{u2} \pmod{p}$$

The reason for the selection of (mod p) based calculation is that, the modular exponentiation in the algorithm 2.1 occurs for (mod p), which is the slowest part in terms of software implementation. Due of this, the (mod q) based modular multiplication operations are separated from modular exponentiation operations of the algorithm. Then the portions of DSA involving modular exponentiation are considered for timing analysis. The timing results are taken using C++ time library, *time.h*, and it is calculated for the unit of seconds. These results are based upon the total execution time of the results of the operations involved. The following table 5.2 gives the results while simulations were done using Pentium 4 (2.6 GHz) processor with 512 MB RAM running Windows XP home edition,

Type of operation	**Time**
Signature operation	1.522 sec
Verification operation	0.531 sec

Table 5.2: Total simulation time for the software blocks of DSA signature and verification operations. The blocks considered for timing analysis were the same as modeled in VHDL for synthesis.

5.2 Hardware Implementation of Digital Signature Standard

The hardware implementation of DSA is mainly composed of the modular exponentiation blocks of the size of prime p. The following portions of DSA algorithm are chosen for hardware implementation.

Signature operation by Alice:

$$\alpha \equiv g^{(p-1)/q} \pmod{p}, \quad \text{then } \alpha^q \equiv 1 \pmod{p} \quad \text{must be satisfied.}$$

$$\beta \equiv \alpha^a \pmod{p}$$

$$r \equiv \alpha^k \pmod{p}$$

Verification operation by Bob:

$$v \equiv \alpha^{u1} \beta^{u2} \pmod{p}$$

The hardware implementation is accordingly divided into the signing and verification units.

5.2.1 Hardware implementation of DSA - Signature Operation

This portion required the implementation of the following sections of the algorithm,

$\alpha \equiv g^{(p-1)/q} \pmod{p}$, then $\alpha^q \equiv 1 \pmod{p}$ must be satisfied.

$\beta \equiv \alpha^a \pmod{p}$

$r \equiv \alpha^k \pmod{p}$

For this, only two Montgomery modular exponentiation units are used. The sequence of the execution of these modular exponentiation blocks is as follows,

- **Two sequential modular exponentiation operations:**

  $\alpha \equiv g^{(p-1)/q} \pmod{p}$, then $\alpha^q \equiv 1 \pmod{p}$ must be satisfied.

It is sequential, because there is a dependency of first generating α and then verifying $\alpha^q \equiv 1 \pmod{p}$. As a result, this requires computational effort of 2 $[(n+1)\ (w+2)]$ clock cycles.

- **Two parallel modular exponentiation operations**:

  $\beta \equiv \alpha^a \pmod{p}$ and $r \equiv \alpha^k \pmod{p}$

The computation of β and r is implemented in parallel. This requires two modular exponentiation blocks to be executed in parallel with computational effort of $(n+1)\ (w+2)$ clock cycles.

The total execution of DSA-Signature operation requires 3 $[(n+1)\ (w+2)]$ units of clock cycles including some additional clock cycles for data I/O. The I/O ports of DSA were designed to fit the FPGA IOBs. That's why; additional load and shift registers were used to load the values in packets by dividing the total number of bits in p, q, and x by the

required port size. i.e. for 1024 bit p, the port size of 32 bits will take 32 parallel load operations. Thus the additional clock cycles required were 2 (w / *port_size*), where w is the number of bits in p. The factor of 2 is used for both input and output data load time required. Figure 5.1 gives the description of how DSA signature block has been implemented in hardware using data produced by the software block. This connection is not the HDL connection, but the pre-calculated numbers generated by the software are used from a text file, "data_file_binary.txt".

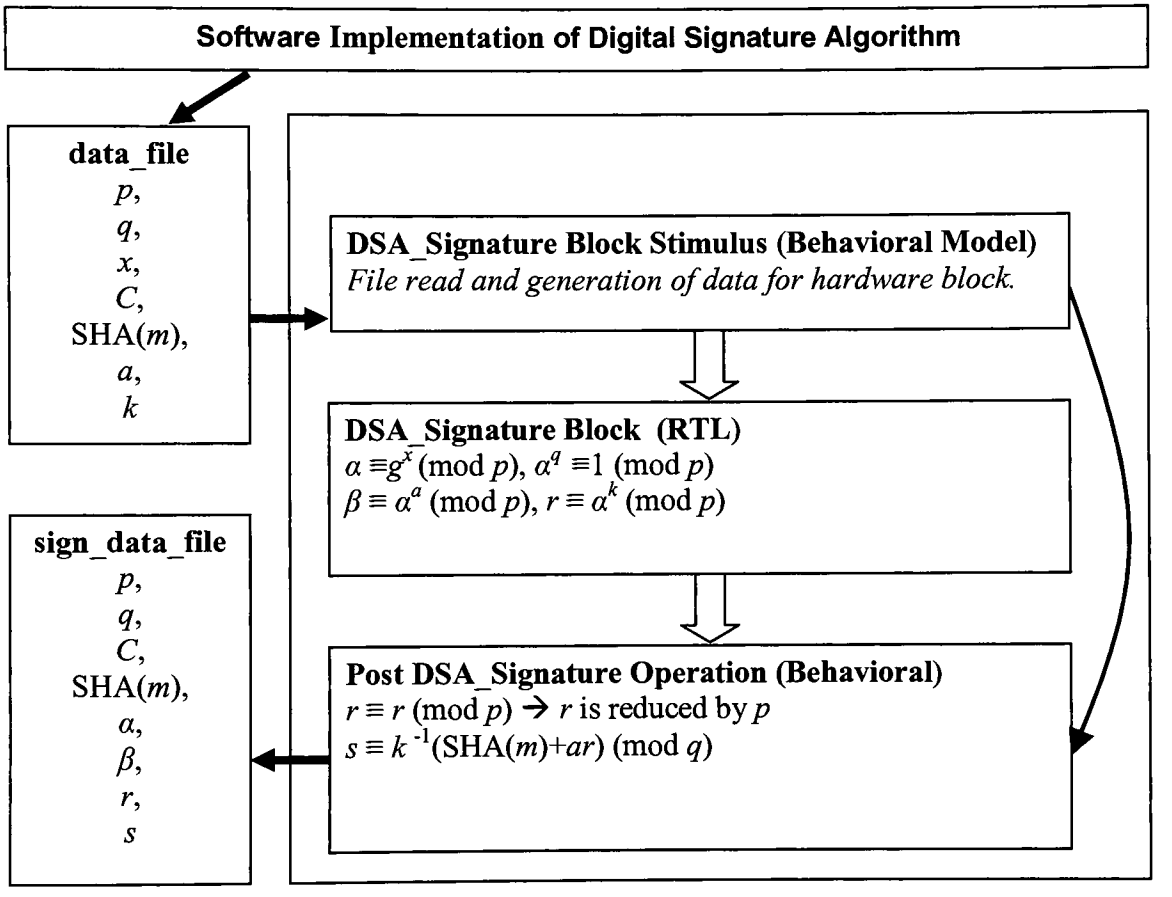
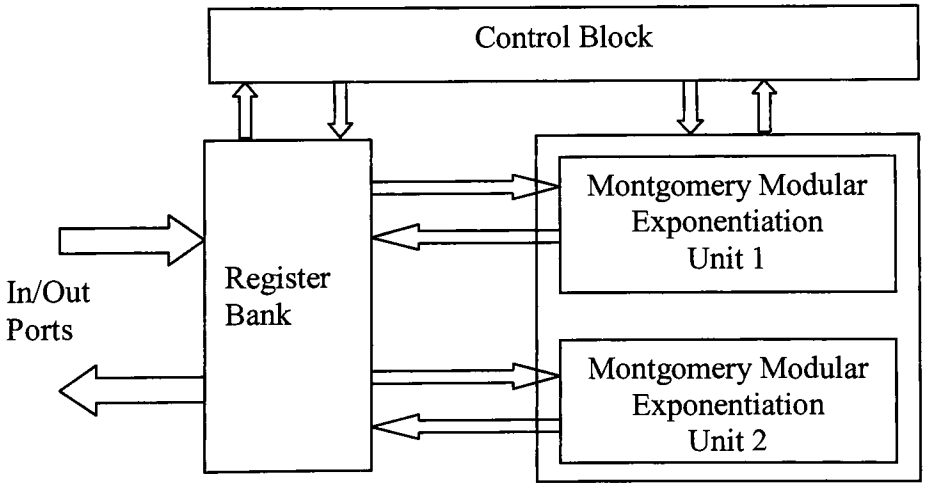


Figure 5.1: Data flow diagram in the hardware Implementation of DSA-Signature block using data produced by the software implemented block.

The hardware implementation for DSA-Signature is only done for those units of DSA algorithm 2.1 where the module exponentiation of the size of p is required. In figure 5.1, the software implementation of DSA performs both the signature creation and its

verification. The file which is created for the hardware block provides the required data to complete the arithmetic operation which involves the modular exponentiation of the size of prime p. The hardware block involves an RTL design which along with a stimulus and post DSA signature block is incorporated in a test bench.

The stimulus reads the data from "data_file_binary.txt" and sends it to the RTL block. The transmission of data is done in shifting small packets, which are then loaded into the RTL block. The RTL block then performs the arithmetic operation as mentioned in figure 5.1, and generates, α, β and r all in the representation of (mod p). These numbers are received by the post DSA-Signature block, which is a behavioral block and performs some post computations as mentioned in figure 5.1. The output from this is then written into a text file, "sign_data_file". The data in "sign_data_file" is then further used by the DSA verification block. The reason behind using the hardware implementation for certain computations but not all is targeting the bottlenecks for speed in the whole design. Therefore the arithmetic portions of DSA, where the speed is not a problem are not implemented in hardware, and this then reduces the cost of implementation as well as the power consumption in the chip.



5.2: Block diagram of DSA Signature Block using two Montgomery modular exponentiation blocks.

In figure 5.2, there are two similar Montgomery modular exponentiation units used. There are eighteen I/O ports used. A description of each port is given according the figure 5.3.

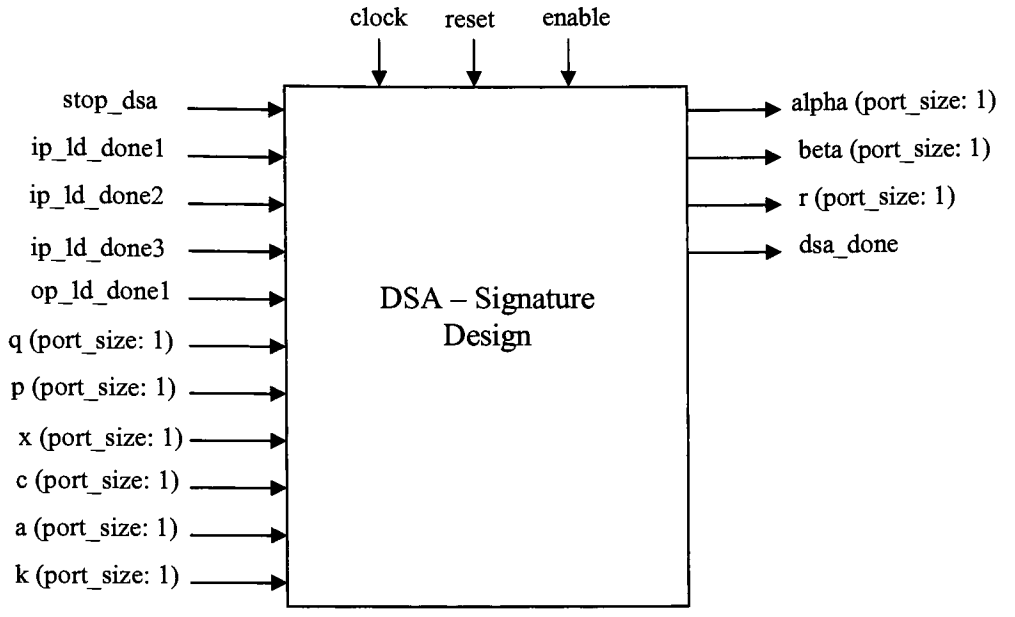


Figure 5.3: Port-level detail of DSA signature block.

The register bank in figure 5.2 has internal registers which store the data sets which are arrived through the input ports shown in figure 5.3. The sizes of these input registers and the corresponding ports are not equal. This is because, for example a 1024 bit design will require a 1024 bit register to store the data sets arriving through port *p*. Similarly for the same design, a 160 bit register will be required to store the data sets arriving from port q. These registers will be explained in the next section. An FPGA has limited number of ports, thus ports of a large design needs to be less then the size of internal data storing registers. The variable *port_size* is a generic variable used for this purpose. The internal registers storing data arriving through these ports varies in sizes, so in order to perform a successful load operation, the internal control signals *ip_ld_done*1, *ip_ld_done*2 and *ip_ld_done*3 are used. For 1024 bit design the *ip_ld_done*1 monitors 1024 bit data to be stored, *ip_ld_done*2 monitors 864 bit data to be stored and *ip_ld_done*3 monitors 160 bit to be stored. Similar to input ports, the output ports are also optimized.

A. Register Bank:

The register bank in figure 5.2 is composed of four different types of shift registers.

a) q_reg, k_sec_reg, a_sec_reg:

These shift registers are identical and store the data sets arrived through input ports of *q*, *k* and *a*. Consider *q_reg*, which is a parallel-in-parallel-out shift register with a control option to load small data sets in parallel. i.e. For a 160 bit size of *q_reg*, the data sets will be equal to *port_size*. Therefore, after storing the first data set it will shift left and then it will keep on loading and shifting new data sets until ip_ld_done3 = '0'. In VHDL it is modeled as follows,

```
q_reg <= q_reg ((q_size - port_size)-1 downto 0) & q;
```

For a 1024 bit DSA design *q_size* = 160 bits and *port_size* = 32 bits. Therefore, on every clock cycle 32 bit data arriving from port *q* is combined on the right side with a previous (*q_size* – *port_size*) data stored in *q_reg*. This forms a left shift register, which is controlled by *ip_ld_done*3 for the number of shifts. i.e. Total of 5 shifts will be required to load 160 bit data. Figure 5.4 describes the operation of a 160 bit shift register.

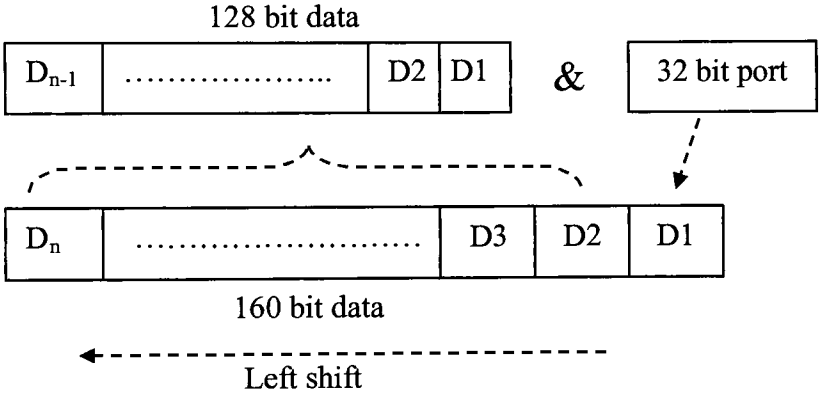


Figure 5.4: Shift register to load data sets in shift left mode.

b) p_reg, p_const_reg:

p_reg works in a similar way as *q_reg* works. It stores the data sets arriving from port *p*. It is of the size of 1024 bits. The number of shifts required to load complete data is controlled by *ip_ld_done*1 control signal. i.e. Total of 32 shifts will be required to complete the load operation of 1024 bit data. *p_const_reg* is similar to *p_reg* and stores data sets arriving through port *c*.

c) p_fact_reg:

p_fact_reg is also of the same type as *p_reg* and *q_reg*. It stores the data sets arriving through port *x*. The size of this register is 864 bits for 1024 bit design. The number of shifts required to completely load the data is controlled by *ip_ld_done2* control signal. i.e. Total of 27 shifts will be required to complete the load operation of 864 bit data.

d) alpha_reg, beta_reg, r_reg:

These registers store the internal results and are connected to the output ports. These are identical and function as load-and-rotate. Load is done when a data set of *port_size* from the left side of these registers is generated through the output ports *alpha*, *beta* and *r*. Rotate is done when it has already sent a *port_size* data set. It rotates left to store the sent data on its right side. The following figure 5.5 explains the operation.

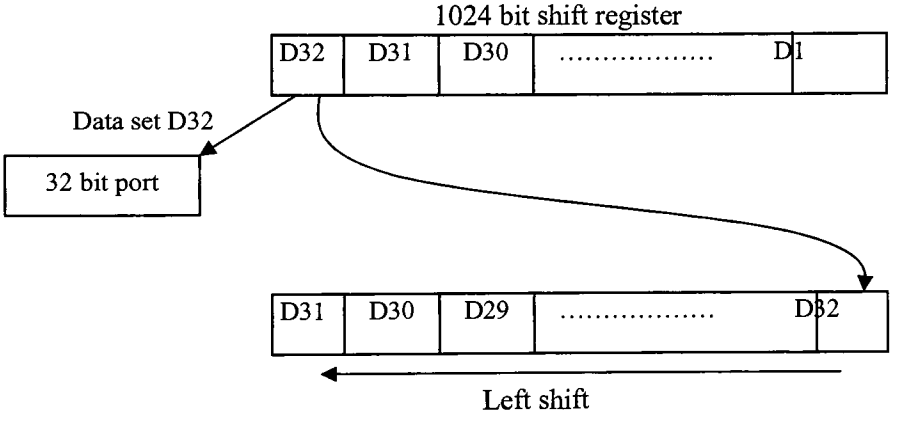


Figure 5.5: Shift register to load data sets in shift left mode.

B. Finite State Machine:

The operation of DSA-Signature block begins, when the input *enable* signal goes high and *stop_dsa* goes low with rising edge of clock cycle. There are four states in which the operation is completed in order to generate final outputs in the form of *alpha*, *beta* and *r*. Initial state is LOAD state, second is PRIMITIVE_ROOT state, third is PUBLIC_KEY state, and fourth is FINAL_OUT state. These states are described according to the state diagram given in figure 5.6.

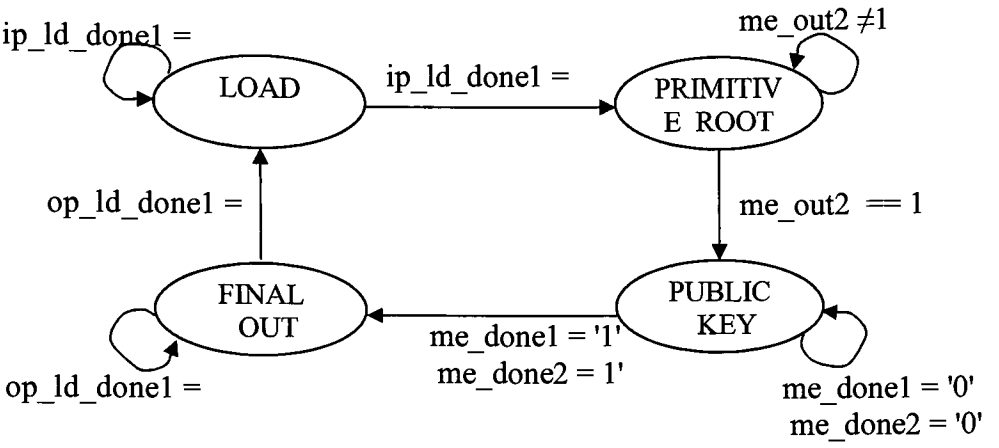


Figure 5.6: Finite Machine of DSA Signature

a) LOAD:

In load state, the registers in the register bank are loaded with data arriving through input ports. FSM remains in LOAD state until the *ip_ld_done*1 is '0'. Once it goes to '1', the FSM makes a transition to PRIMITIVE_ROOT state.

b) PRIMITVE_ROOT:

In this state the operation of finding g as primitive root (mod p) is computed. For this purpose the following modular exponentiation operations are performed sequentially. First $\alpha \equiv g^{(p-1)/q} \pmod{p}$ is computed after the selection of g. After that, α^q is computed to be verified as $\alpha^q \pmod{p} \equiv 1 \pmod{p}$. This requires random number of attempts in verifying $\alpha^q \equiv 1 \pmod{p}$ based upon the selection of prime numbers p and q. The output of produced in this state is α. This operation requires $2(n+1)$ $(w+2)$ number of clock cycles. Once the output is produced, the FSM makes a transition of PUBLIC_KEY state.

c) PUBLIC_KEY:

In this state, *beta* and r are computed as, $\beta \equiv \alpha^a \pmod{p}$ and $r \equiv \alpha^k \pmod{p}$. This operation requires two Montgomery modular exponentiation blocks to be executed in parallel. Therefore, the total time of computation is $(n+1)(w+2)$ clock cycles. Once the values of β and r are computed, the FSM makes a transition to FINAL_OUT state.

d) FINAL_OUT:

In this state, the values of α, β and r are generated as output. The FSM remains in this state until the *op_ld_done*1 is '0', and once it goes to '1', the FSM makes a transition to load state.

5.2.1.1 Simulation results for DSA-Signature block

Three different sizes of DSA-Signature are simulated at RTL and post-synthesis levels. The first two simulations are done for 12 and 32 bits designs and the third simulation is done for 1024 bit design. The reason for the simulations of 12 and 32 bit designs is due to the verification of design accuracy at the earlier design phases. A simulation at post-synthesis level for a 32 bit design took approximately two hours on a single AMD 64 bit computer, while a 1024 bit design took approximately seven days on the same configuration of system. For post synthesis simulation the Modelsim simulator on AMD 64 bit computer was not optimized to simulate the gate level model of DSA generated by Xilinx ISE FPGA. This required the simprim.lib library to be configured for Modelsim. simprim.lib provides the gate level components for post synthesis simulation. Instead of 160 message digest SHA(m), a random message m of size 12 bit is used for 12 and 32 bit designs. Considering the three design sizes and their simulation results as follows,

DSA-Signature Designs of 12 and 32 bit sizes:

Consider figure 5.1 as a reference for the flow of design. The sizes of the variables in 12 bit design are p, c = 12 bit; q, a, k, m, x = 6 bit. Similarly the variable sizes of the 32 bit design are p, c = 32 bit; q, a, k, m = 12 bit; x = 20 bit. The input to this design is according to the table 5.3,

Data	Value			
	12 bit Design	**Size in bits**	**32 bit Design**	**Size in bits**
p	2543	12	129023	32
q	41	6	2081	12
x	62	6	62	20
a	3	6	38	12
k	28	6	1824	12
m	13	6	3277	12
c	1126	12	12080	32

Table 5.3: Data input for 12 and 32 bit hardware block of DSA Signature.

This design was simulated at RTL and post-synthesis level using Xilinx ISE and Modelsim for Vertex2p – xc2vp100-5ff1696 FPGA. The output achieved from the hardware block is fed into the post DSA-Signature block, which further computes $s \equiv k^{-1} (m + ar) \pmod{q}$ and creates the result text file "sign_data_file". This block is not synthesizable.

Figure 5.7 shows the 12 bit DSA-Signature design wave form for the hardware block. This simulation is done at RTL level; therefore, a minimum of 2 ns clock cycle is used. The selection of 12 bit design has been given in order to display and explain the results. This wave form shows only the initial load operation. Notice the input and output data ports, labeled as *p*, *q*, *x*, *c*, *a*, *k*, *alpha*, *beta*, and *r* are of the size of *post_size*, which is 2 bit in this case. Total number of 6 clock cycles is required for 2 bit sized ports to load all the data sets arriving at the input ports. A difference 6 clock cycles has been shown in the wave form between the point, when *dsa_state* = LOAD and when *ip_ld_done*1 = '1'.

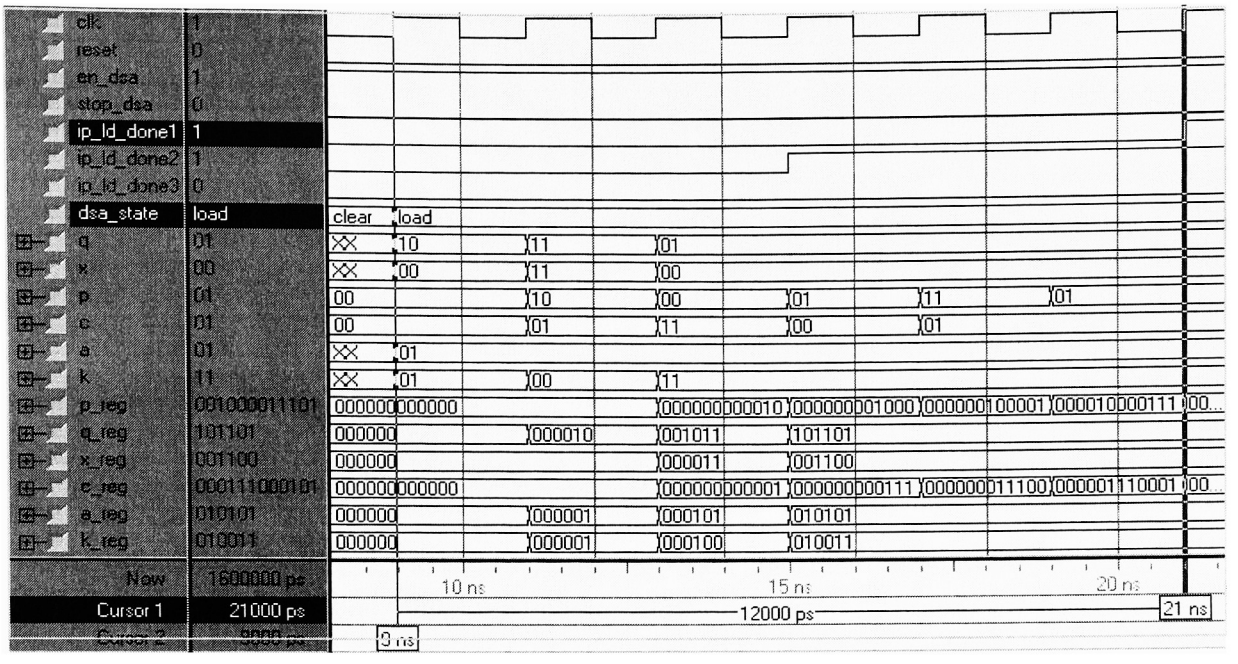


Figure 5.7: Wave form simulation for 12 bit DSA signature design showing the load operation completed in 6 clock cycles.

Table 5.4 gives the values of α, β, r and s, where α, β are part of the public key, and r and s represents the signed message.

Data	**Value**			
	12 bit Design	**Size in bits**	**32 bit Design**	**Size in bits**
α	817	12	96956	32
β	2335	12	47605	32
r	1	6	1092	12
s	24	6	676	12

Table 5.4: Data produced from the DSA Signature block for DSA verification block.

The next wave form in figure 5.8 shows the completion of DSA signature block operation, and therefore in the last state of FSM which is FINAL_OUT, it generates *alpha*, *beta* and r for the *post_dsa_block*. The *post_dsa_block* computes the operations in order to produce the final signatures, which are *r_sign* and *s_sign* shown in the wave form. *r_sign* and *s_sign* are equal to r and s as shown in the table 5.4.

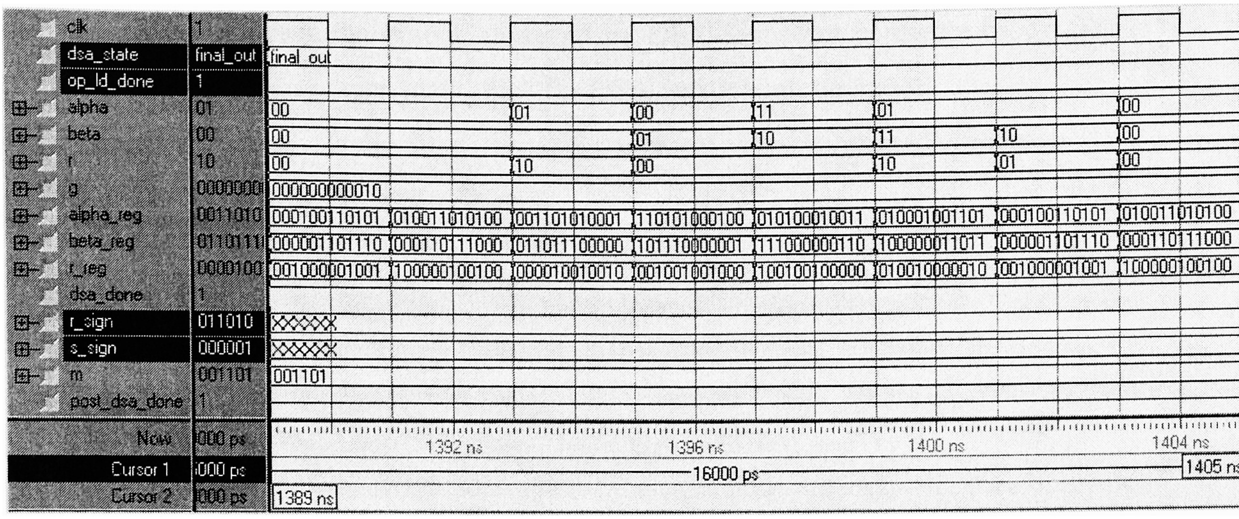


Figure 5.8: Output generated at 1405 ns for 12 bit DSA signature design using 2 ns clock. Wave form shows the final output operation.

DSA-Signature Design of 1024 bit size:

The simulations of 1024 bit design include 160 bit SHA(*m*) message digest. The data input for the hardware block is given in table 5.5.

Data	**Size in Bits**	**Value (Hexadecimal)**
p	1024	0x040000000000000000000000000000000000cef8000000000000000000000 00000000000000000000000000000000000000000000040000000000000000 0000000000000000000cef800000000000000000000000000000000000000000 00000000000000000000000000000000000000000000000000000000000001
q	160	0x08000000000000000000000000000000000019df
x	864	0x0800000000000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000080000000000000000000 00000000000000000000000000000000000000000000000000000000000000 0000000000000000000000
a	160	0x06000000000000000000000000000000000019df
k	160	0x07000000000000000000000000000000000019df
SHA(*m*)	160	0x0c3d2e1f09032547618badcfe427656d1a26e04b8
c	1024	0x0208a789a4585906824f7080000000000a8d29256ffffffffffffffffffffffffeffffd62af bf0000000000000000000000000080a7d2371d8385906824f7080000000000a8b8 97ff4dcc753dfc00000000000000000002a2d7049686e5a2caf7ffffffffffffffbf779a 4e809b682222bdf3f00000000000000d0c12

Table 5.5: Data input produced by software for 1024 bit hardware block of DSA Signature.

Table 5.6 shows the output produced by DSA-Signature hardware block. These results are then stored in "sign_data_file" for DSA verification operation.

Data	**Size in bits**	**Value (Hexadecimal)**
α	1024	0x03558df42ac52a3c7775f2000ebe4f9bbe81093a4cc3e8b19ca38d82e77e1a522f9888 1adab295f75e1de9e4b48f24503a81b38af99e55ed2db427c24c8f30cf3b3a4640d45222 0ddc066c0c2155137d38e6c1b4b42d258cbe23e9279eb7350e80a209ab34de7c9779b10 d4a9612313d675622a9d8e475df185e18018f8ff1fa4
β	1024	0x01f5e6add9f32975859c942f09152d96a2673a20ec46216a5148fbf37bee5500190368 62295aa8417bddfd7f254cfa87b1e646e6d88b9e80dc34574318bc90126f2165b594513d e402639477e932ef1c9f5d788f9596c738545978ef37cf772aef8452f367c19487a6513d7 6cfa7e43796810c156b810af3f20c940558ebedd27
r	160	0x048e275882ab7c8a22efa4e19bc298fa4face7de8
s	160	0x0195e82962bc98918e8ac48aafbc981574704f5ae

Table 5.6: Data produced from the DSA Signature block for DSA verification block.

Total number of 3164269 clock cycles is required to generate the output given in table 5.6. The number of clock cycles in 12, 32, and 1024 bit designs depends upon various factors. The first thing to consider here is finding g as primitive root, when the condition of $\alpha^q \equiv 1 \pmod{p}$ is true. The number of clock cycles used to satisfy this condition varies for various selections of prime numbers p, q and even number x. Consider the best case, when for $g = 2$, $\alpha \equiv g^{\lambda} \pmod{p}$ satisfies the condition of $\alpha^q \equiv 1 \pmod{p}$, then this will require execution of two sequential modular exponentiation. For the best case, it will require 2[(n+1)(w+2)] clock cycles. Computation of β and r for (mod p) requires two parallel modular exponentiations at the computational effort of $(n+1)(w+2)$ clock cycles. As a result, total of $3[(n+1)(w+2)]$ clock cycles will be required for a best case implementation of DSA-Signature hardware block.

5.2.1.2 Synthesis results for DSA Signature block

The design was synthesized for 12, 32 and 1024 bit sizes. Table 5.7 gives the synthesis using Vertex2p family xc2vp100 – 5ff1696 FPGA. Total number of slices available on Vertex2p FPGA was 44096 and total number of IOBs was 1164.

Size of Design (bits)	Port Size	No. of Slices	Clock Period (ns)	No. of IOBs
12	2	621	8.2	26
32	4	1274	8.7	26
1024	32	38675	52.3	296

Table 5.7: Synthesis results taken for 12, 32 and 1024 bit DSA Signature designs.

5.2.1.3 Comparison between 1024 bit hardware and its equivalent software design

The execution of software design required 0.14 seconds to perform the DSA-Signature operation for equivalent arithmetic portions. Consider table 5.8 for the comparison between hardware and software implementations of DSA-Signature block.

Simulation, HW & SW	Time
Hardware, using 53 ns clock cycle	3164269 * 53 = 167706257ns = 0.167 seconds
Software for signature equivalent block	1.522 seconds
Difference	1.355 seconds

Table 5.8: Comparison between hardware and software implementation in terms of speed.

As seen from table 5.8, the software block required 1.355 seconds more time to perform the operation. This comparison is based upon:

$$\text{Speed Improvement} = \frac{\text{Time taken to complete operation in Software}}{\text{Time taken to complete operation in Hardware}}$$

$$\text{Speed Improvement} = \frac{1.522}{0.167} = 9.11 \text{ times}$$

Therefore, the hardware implementation of DSA signature block is 9.11 times faster then software. The speedup attained in hardware is caused by some factors given below in terms of the arithmetic operations performed in both hardware and software.

- $\beta \equiv \alpha^{a} \pmod{p}$ **and** $r \equiv \alpha^{k} \pmod{p}$

The execution of β and r is sequential in software which requires $2[(n+1)(w+2)]$ clock cycles. On other hand, parallel execution in hardware requires $(n+1)(w+2)]$ which provides significant speedup when compared to software. The other factors which cause the slow speed in software are execution of 1024 bit data on 32 bit processor and longer interconnects. The next section shows the implementation, execution and synthesis results of DSA-Verification block.

5.2.2 Hardware implementation of DSA-Verification Operation

DSA verification block involves the following set of operations;

1. $u_1 \equiv s^{-1}\, \mathrm{SHA}(m) \pmod{q}$ and $u_2 \equiv s^{-1}\, r \pmod{q}$.
2. $v1 \equiv \alpha^{u1} \pmod{p}$
3. $v2 \equiv \beta^{u2} \pmod{p}$
4. $v \equiv v1\ v2 \pmod{p} \pmod{q}$
3. Finally, comparing, if $v = r$

i.e. if this is true, the signature is valid, otherwise not.

Among these set of arithmetic operations, $v1 \equiv \alpha^{u1} \pmod{p}$ and $v2 \equiv \beta^{u2} \pmod{p}$ are implemented in hardware.

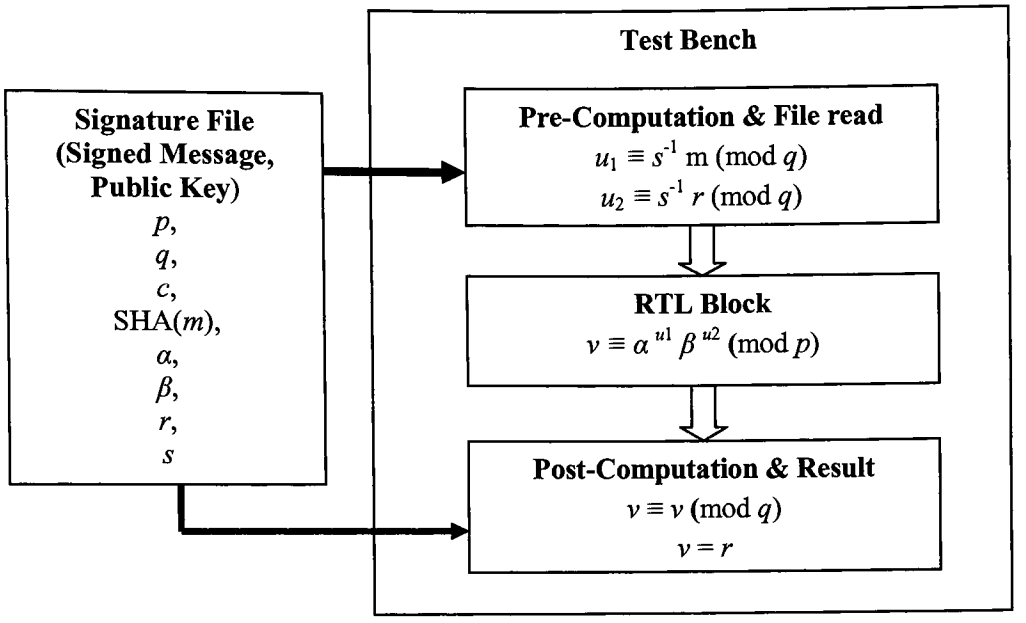


Figure 5.9: Data flow of hardware implementation of DSA verification block.

Figure 5.9 shows the implementation in terms of behavioral and RTL partitioning of the DSA verification block. Data inputs are read from the text file "sign_data_file", which is produced by DSA-Signature block.

Initially, $u1 \equiv s^{-1}$ SHA(m) (mod q) and $u2 \equiv s^{-1}r$ (mod q) are computed behaviorally outside of the RTL block. This requires use of Extended Euclidean Algorithm for finding a modular multiplicative inverse of s^{-1}. Once $u1$ and $u2$ are computed, they are sent to the RTL block using parallel data shift operation as mentioned in DSA-Signature block. In the RTL block,

$$v_1 \equiv \alpha^{u1} \pmod{p}$$

and,

$$v2 \equiv \beta^{u2} \pmod{p}$$

are computed. Then, $v1$ and $v2$ are sent to the post-compute block of DSA-Verification. In the post-compute block, $v \equiv (v1v2 \pmod{p}) \pmod{q}$ is computed. Finally, the v is compared with r. r is generated by DSA-Signature block If $v = r$, then the signature is valid, otherwise it is not. In this case the RTL block requires two Montgomery exponentiation units to be instantiated, and executed in parallel. The computational effort

is $(n+1)(w+2)$ clock cycle. The block diagram of the RTL module is shown in figure 5.10.

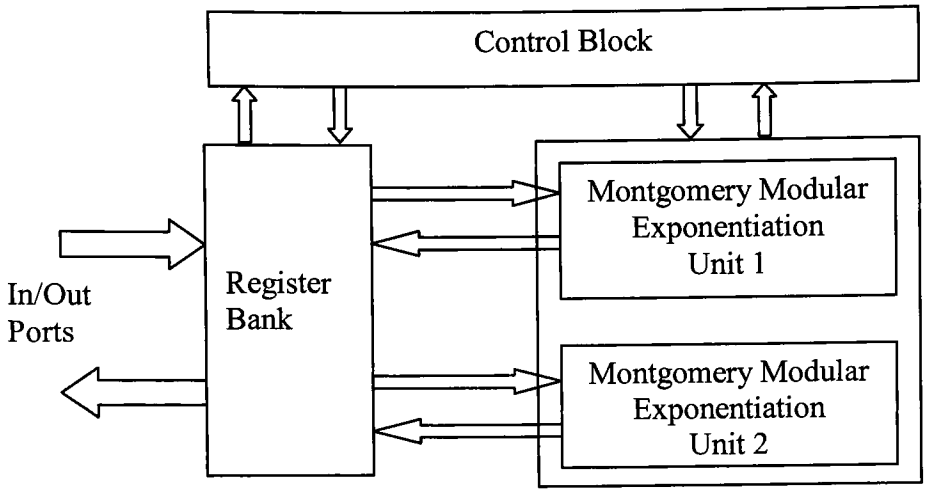


Figure 5.10: Block diagram of DSA Verification Block using two Montgomery modular exponentiation blocks.

In figure 5.10, two Montgomery modular exponentiation blocks are used in addition to internal registers and a control block. Figure 5.11 shows the port level detail of DSA-Verification block.

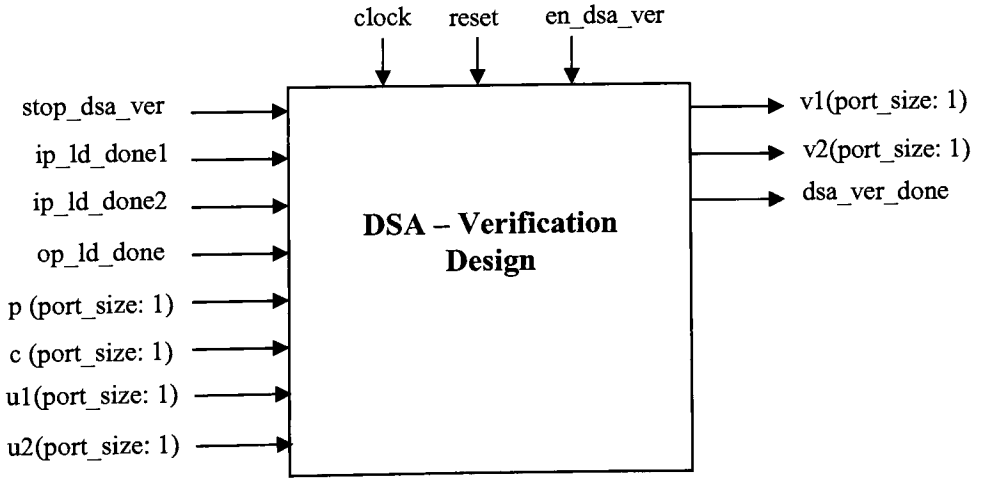


Figure 5.11: Port-level detail of DSA verification block.

The explanation of the register bank and the control block in figure 5.10 is given below.

A. Register Bank:

The register bank in figure 5.10 is consisted of internal registers. The structure of loading and unloading the data is similar to the registers used in the register bank of DSA signature block.

a) p_reg, p_const_reg, alpha_reg, beta_reg:

These registers are identical and their size is equal to the prime number p. Data sets stored are p, c, α and β. The input to these registers is stored by shifting small data sets to the left. Consider figure 5.4 for the operation.

b) u1_reg, u2_reg:

These registers are identical and their size is equal to the prime number q. They store data arriving through ports $u1$ and $u2$ respectively. The structure of these registers is same as shown in figure 5.4.

c) v1_reg, v2_reg:

These registers are identical and their size is equal to the prime number p. The output is first stored in these registers and then generated through the output ports $v1$ and $v2$. The structure of these registers is same as shown in figure 5.5.

B. Finite State Machine:

The state machine is composed of four states. There are two load input ports and output ports states, where data transactions are done. One state is used to compute the modular exponentiation operations, and one state is used to clear all the registers. The design is activated on low reset. Each state is explained below:

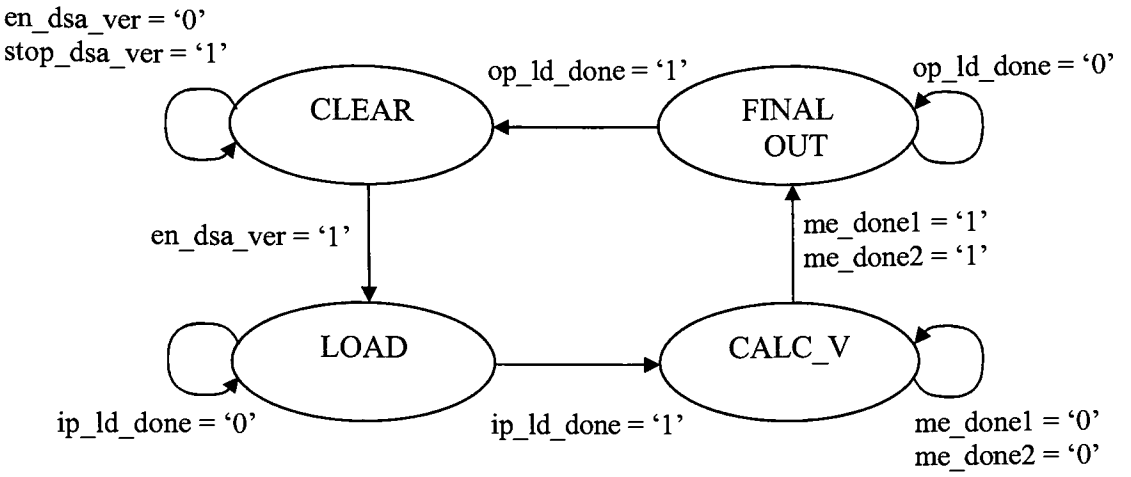


Figure 5.12: Finite state machine for DSA verification block.

a) CLEAR:

FSM remains in clear state as long as *en_dsa_ver* = '0' and *stop_dsa_ver* = '1'. Initially, *stop_dsa_ver* = '0'. This condition is also required at the end of the design when *stop_dsa_ver* is asserted to logic '1' to stop the DSA block and then the FSM enters the CLEAR state. From CLEAR state, the FSM makes transition to LOAD state, when *en_dsa_ver* = '1'. During this transition, FSM clears all the registers.

b) LOAD:

In LOAD state, the FSM loads the input registers until the *ip_ld_done*1 = '0'. *ip_ld_done*1 is controlled by the pre-computation block, which also acts as generator. The input registers are loaded with values and then FSM changes the state to CALC_V when *ip_ld_done*1 = '1'.

c) CALC_V:

In this state, the FSM sets the values for the two Montgomery exponentiation blocks, and remains in CALC_V state until *me_done*1 and *me_done*2 are low. FSM makes transition to FINAL_OUT state, when both *me_done*1 and *me_done*2 are asserted to logic '1'. During this transition, the outputs of the Montgomery modular exponentiation are generated as *v*1 and *v*2.

d) FINAL_OUT:

In this state, the data is generated out using the output shift registers. The FSM remains in this state until *op_ld_done* = '0'. Once *op_ld_done* = '1', the FSM makes transition to CLEAR state. At this time, the signal *stop_dsa_ver* is asserted to '1', and as a result the FSM remains in CLEAR state without making any transition to the LOAD state.

5.2.2.1 Simulation results for DSA verification block

The DSA-Verification block is executed for 12, 32 and 1024 bits of design sizes. The 160 message digest SHA(m) is only used for 1024 bit design while a random message m is used for 12 and 32 bit designs.

DSA-Verification Designs of 12 and 32 bit sizes:

Consider 5.9 as the inputs to these designs. These inputs are same as produced by the DSA signature block.

Data	**Value**			
	Size in bits	**12 bit Design**	**Size in bits**	**32 bit Design**
q	6	41	12	2081
m	6	13	12	3277
p	12	2543	32	129023
c	12	1126	32	12080
α	12	817	32	96956
β	12	2335	32	47605
r	6	1	12	1092
s	6	24	12	676

Table 5.9: Inputs to DSA verification block.

Figure 5.13 shows the waveform simulations for the 12 bit DSA verification design. It displays the initial load operation of the internal registers. The data inputs are loaded in total number of 6 clock cycles when *ip_ld_done* = '1'. The input ports shown

are p, c, *alpha*, *beta*, $u1$ and $u2$. $u1$ and $u2$ are computed in the behavioral VHDL model prior to loading data to the RTL block.

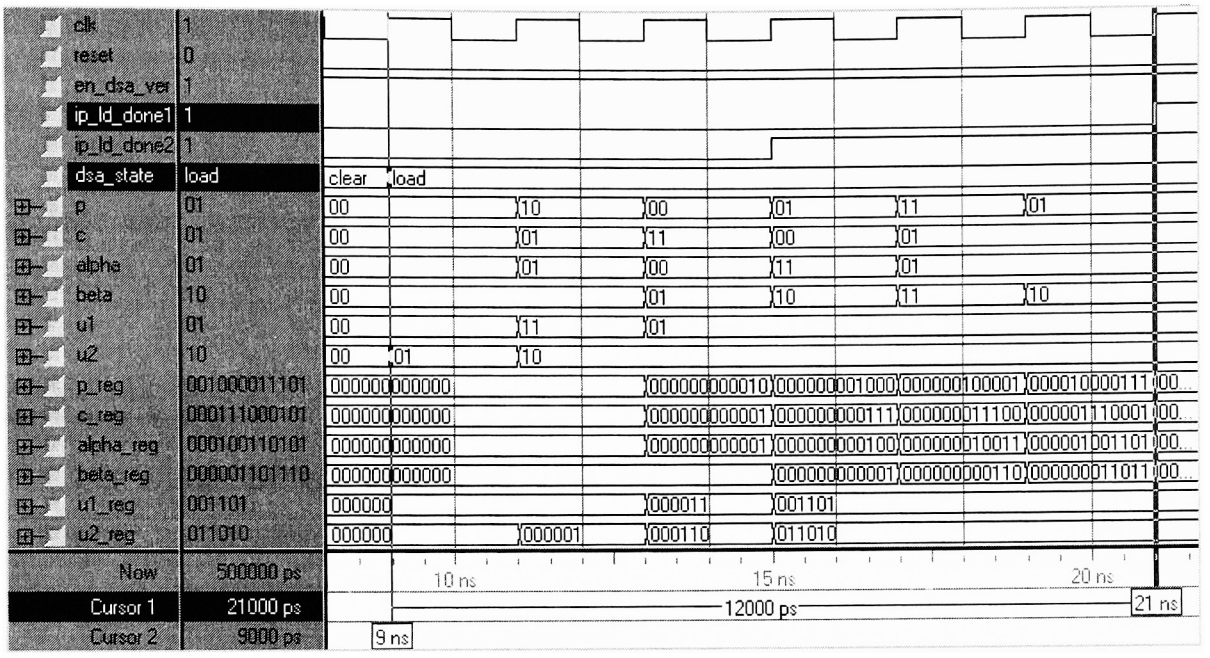


Figure 5.13: Wave form simulations for the load operation of DSA verification block.

Consider table 5.10 for the outputs generated from pre-computation, RTL and the post computation blocks.

Block	**Data**				
		bits	**12 bit Design**	**bits**	**32 bit Design**
Pre-computation	$u1$	6	33	12	482
Pre-computation	$u2$	6	12	12	802
RTL	$v1$	12	1600	32	111437
RTL	$v2$	12	333	32	55018
Post-computation	v	6	1	12	1092
DSA Signature	r	6	1	12	1092

Table 5.10: Outputs from DSA verification block. r is added to this table to give comparison with v.

Figure 5.14 displays the results of the 12 bit design obtained in table 5.10 .

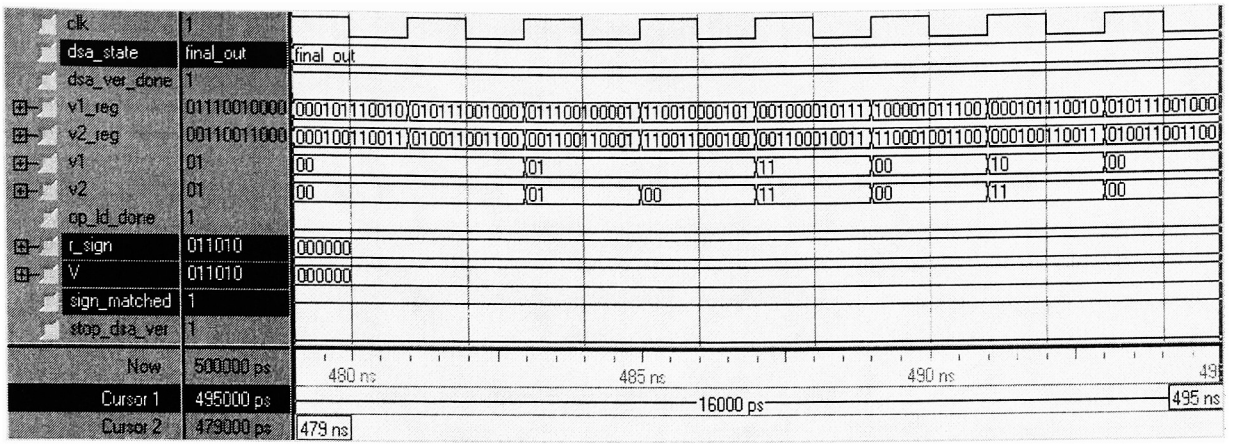


Figure 5.14: Wave form simulations for the final out put of 12 bit DSA verification block.

DSA-Verification Design of 1024 bit size:

Consider table 5.11 for the inputs given to the DSA verification block.

Data	Value (Hexadecimal)
q	0x0800000000000000000000000000000000019df
m	0x0c3d2e1f09032547618badcfe427656d1a26e04b8
p	0x040000000000000000000000000000000000cef80000000000000000000000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000000000000cef800

Table 5.11. Input values for DSA verification block.

Consider table 5.12 for the outputs generated from pre-computation block, RTL and the post computation block.

Block	Data	Values
Pre-computation	$u1$	0x0505e9985bcfc4c8c420d4bbda5803c02839a695c
Pre-computation	$u2$	0x076f9691523cb6f14e76a53ad52807ba767c82f7f
RTL	$v1$	0x01346abe0d317f1a8414b9788ad547544c0356f7ead29fbaa7b981a2a8a4b628c121020693d9b5b56231e1229b0afbd841ca1f86f446ddcd099187cf708e8488d7b030bfc396e4a58ac6a9a678b09a607faa869c98542ac36ef1a6b81b40c7eb88e02cb51c5e8c3ad744271d633889c6b4dd4dbfcdd12c29d7dbaf9095ae859c7
RTL	$v2$	0x01ca27e9e400aafa23a537891a431462ab9f4b85a680c5dd3cc719596662007ab97f80e15686cbe1089faea0bf61857ec096791d20134d23dfe6651b91319aa3c4072a2584f520a366dad2e5346474b14b27e30530068b1be0be027c21c4bb75311491949662087d21916d4987795685dd65d9f9dcc1760064abd0800511ee445
Post-computation	v	0x048e275882ab7c8a22efa4e19bc298fa4face7de8
DSA Signature	r	0x048e275882ab7c8a22efa4e19bc298fa4face7de8

Table 5.12. Output values from DSA verification block except r, which is given here for comparison.

Total number of 1054804 clock cycles is required to complete the 1024 bit DSA-Verification operation in hardware. For a 1024 bit Montgomery modular exponentiation, the time period required is $(n+1)(w+2)$, where $n = w+2$ and $w = 1024$. So 1053702 clock cycles are consumed by the modular exponentiation block and 1102 clock cycles are required for the loading the data at input and output ports plus the internal state transitions.

5.2.2.2 Synthesis results for DSA verification block

The RTL block of DSA-Verification is synthesized for 12, 32, and 1024 bits. Table 5.13 shows the synthesis results using Vertex2p family xc2vp100 – 5ff1696 FPGA

Size of Design (bits)	Port Size	No. of Slices	Clock Freq (ns)	No. of IOBs
12	2	555	8.2	23
32	4	1160	8.7	23
1024	32	34174	52.3	263

Table 5.13: Synthesis results taken for 12, 32 and 1024 bit DSA verification blocks.

5.2.2.3 Comparison between 1024 bit hardware and its equivalent software design

The software execution of DSA-Verification took 0.531 to complete the operation. Consider table 5.14 for the comparison of time taken to complete the DSA verification block implemented in hardware as well as in software.

Simulation, HW & SW	Time
Hardware, using 53 ns clock cycle	1054804 x 53 = 55904612 ns = 0.055 seconds
Software for signature equivalent block	0.531 seconds
Difference	0.476 seconds

Table 5.14. Comparison between hardware and software implementation in terms of speed.

As seen from table 5.8, the software block required 0.476 seconds more time to perform the operation. This comparison is based upon:

$$\text{Speed Improvement} = \frac{\text{Time taken to complete operation in Software}}{\text{Time taken to complete operation in Hardware}}$$

$$\text{Speed Improvement} = \frac{0.531}{0.055} = 9.65 \text{ times}$$

Therefore, the hardware implementation of DSA signature block is 9.65 times faster then software. The speedup attained in hardware is caused by some factors given below in terms of the arithmetic operations performed in both hardware and software.

- $v_1 \equiv \alpha^{u1} \pmod{p}$ **and,** $v2 \equiv \beta^{u2} \pmod{p}$

The execution of v_1 and v2 is sequential in software and requires $2[(n+1)(w+2)]$ clock cycles. On other hand, parallel execution in hardware requires $(n+1)(w+2)]$ which provides speedup when compared to software. The other factors which cause the slow speed in software are execution of 1024 bit data on 32 bit processor and longer interconnects.

5.3 Summary

The implementation of Digital Signature Algorithm in hardware and software has been given for speed comparison. The execution of the multi-precision DSA for 1024 bits is very slow in software as compared to hardware. This is due to two major reasons; One is the general purpose processor process 32 bit data for 1024 bit DSA operation which requires serial arithmetic logic unit operation. Second reason is the sequential execution of parallel portions of algorithms. On the other hand, in hardware, these two problems are resolved by having 1024 bit parallel data operation as well as parallelism for non data dependent blocks.

Chapter 6
Conclusions and future work

In this thesis, the implementation of multi-precision modular arithmetic was done because of its importance for public key cryptosystems. An algorithm of Montgomery modular multiplication was implemented in two different hardware designs and upon comparison of the architectures and performance, the faster design was selected for further work. This faster design was then compared with 1024 bit software based modular multiplication which resulted in low performance caused by the hardware block. An analysis of left-to-right and right-to-left Montgomery modular exponentiation was given and to achieve more speed in hardware, right-to-left Montgomery exponentiation algorithm was chosen for the implementation in hardware. It was implemented using the fast architecture of Montgomery modular multiplication. Modular exponentiation block was designed as an internal part of any public key cryptosystem and thus Digital Signature Algorithm was selected to implement this block. Current standard of 1024 bit DSA design was implemented. It contained two design units, DSA-Signature unit and DSA-Verification unit. These designs were targeted only for those portions of DSA where the bottlenecks for the speed were present. Thus a combination of RTL and behavioral designs were implemented. A software version of DSA was also implemented using SystemC and this implementation was at algorithmic level. The CPU time for the software implemented DSA was noted and compared with the simulation time of hardware DSA. Only those portions of software implementation were exposed to timing analysis which were also implemented in hardware. It was shown that DSA signature unit was 9.11 times faster and DSA verification unit was 9.65 times faster then the software units.

The objective of this thesis research was to recognize the speed bottlenecks in the software implementation of public key cryptosystems and removing them by hardware implementation. The speedup obtained in hardware was approximately 10 times faster then software. This speedup can be further improved by considering some of the following factors.

a) Selection of faster technology libraries to implement Montgomery modular exponentiation. i.e. Standard cell ASIC.

b) Introducing further partitioning and pipelining to increase the throughput.

c) A modification in the right-to-left Montgomery modular exponentiation block can save some clock cycles. Consider the following lines of pseudo code of algorithm for Right-to_left Montgomery modular exponentiation.

4. for $i = 0$ to w-1 loop
   4.1 if (E (i) = 1) then
      4.1.1 H = Monpro(H, P, M) ---- Multiply
   4.2 P = Monpro (P, P, M) ---- Square
5. End for.
6. H = Monpro (1, H, M)

This loop runs for w times, which is the number of bits in M. The purpose of loop is to perform multiply and square operations. This block will remain affective until the last E (i) = '1' is available. After that, it will not be affective for all the bits in E until the MSB arrives, which is also '0'. Consider a small example,

$P = 1001$

$M = 1011$

$E = 0011 = \{E(3), E(2), E(1), E(0)\}$.

Then in this case, the block will update the line 4.1.1 until E (1), E (0) are arrived. After that for E (3) and E (2), this block will not do any useful work, because at line 6 the data needed from block is H. This will waste extra two clock cycles. A suitable modification of this algorithm may remove this problem.

d) Besides DSA, Montgomery modular exponentiation block can also be implemented for RSA and Diffie-Hellman key exchange scheme. This will provide further speed comparison with software based implementations. A modified version of Montgomery modular exponentiation algorithm can be used for Elliptic Curve cryptosystems.

References

1. Cryptography – [Online]
Available: http://www.oft.state.ny.us/esra/Guidelines_files/ESRAGuidelines5.htm

2. Stream ciphers – [Online]
Available: http://www.ssh.com/support/cryptography/algorithms/symmetric.html

3. Wade Trappe, and Lawrence C. Washington, "Introduction to Cryptography with coding theory", *Prentice Hall*, 2002.

4. Manindra Agarwal, Nitin Saxena and Neeraj Kayal. "Primes is in P", Preprint, Aug. 6, 2002.
Available: http://www.cse.iitk.ac.in/primality.pdf

5. Peter L. Montgomery, "Modular Multiplication without Trial Division", published in *Mathematics of Computation*, Volume 44. Number 170, April 1985. Pages 519-521.

6. Alan Daly, Willian Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *International Symposium on Field Programmable Gate Arrays*, February 24-26, 2002, Monterey, California, USA.

7. Secure Hash Standard, SHA-1, http://www.itl.nist.gov/fipspubs/fip180-1.htm

8. Thomas Blum and Christof Paar, "Montgomery modular exponentiation on reconfigurable hardware", *14th IEEE Symposium on Computer Arithmetic* (ARITH-14) April 14-16, Adelaide, Australia.

9. Nadia Nedjah and Luiza de Macedo Mourelle, "Reconfigurable Hardware Implementation of Montgomery Modular Multiplication and Parallel Binary

Exponentiation", in proceedings of *Euromicro Symposium on Digital System Design* (DSD'02) 0-7695-1790-0/02, 2002 IEEE.

10. Daniel M. Gordon, "A survey of fast exponentiation methods", in *Journal of Algorithms* 27, 129-146(1998), Article No. AL970913.

11. Douglas Stinson , "Cryptography, theory and practice", Chapman & Hall/CRC; 2nd edition, 2002.

12. Donald Ervin Knuth, "The Art of Computer Programming", Volume 2: Semi-numerical Algorithms. Reading, Massachusetts: Addison-Wesley, 2nd edition, 1981.

13. S. B. Ors, L. Batina, B. Preneel, J. Vandewalle, "Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array", *Parallel and Distributed Processing Symposium*, 2003. Proceedings International, 22-26 April 2003 Page(s):8 pp.

14. Taher Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *Information Theory, IEEE Transactions on Volume 31*, Issue 4, July 1985 Page(s):469 – 472.

15. Eun-Jun Yoon, Eun-Kyung Ryu, and Kee-Young Yoo, "Efficient Remote User Authentication Scheme based Generalized ElGamal Signature Scheme", *IEEE Transactions on Consumer Electronics*, Vol. 50, No. 2, MAY 2004.

16. Harn, L.; Xu, Y. "Design of generalised ElGamal type digital signature schemes based on discrete logarithm", *Electronics Letters Volume 30*, Issue 24, 24 Nov. 1994 Page(s):2025 - 2026

17. Digital Signature Standard – [Online]
Available: http://www.itl.nist.gov/fipspubs/fip186.htm.

18. P. Kitsos, N. Sklavos and O. Koufopavlou, "An efficient implementation of the digital Signature algorithm", *9th International Conference on Electronics, Circuits and Systems*, 2002. Volume 3, 15-18 Sept. 2002 Page(s):1151 - 1154 vol.3.

19. G. Joseph.; W.T. Penzhorn, , "High-speed algorithms for public-key cryptosystems", AFRICON, 2004. *7th AFRICON Conference in Africa Volume* 2, 15-17 Sept. 2004 Page(s):945 - 951 Vol.2